Spatial Computation

Mihai Budiu

December 2003

CMU-CS-03-217

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy*

Thesis Committee:
Seth Copen Goldstein, Chair
Peter Lee
Todd Mowry
Babak Falsafi, ECE
Nevin Heinze, Agere Systems

This research was sponsored by the Defense Advanced Projects Agency (DARPA) under US Army contract no. DABT6396C0083, DARPA under US Navy grant no. N000140110659 via Pennsylvania State University and via Hewlett-Packard (HP) Laboratories, the National Science Foundation (NSF) under grant no. CCR-0205523 and no. CCR-9876248, and by generous donations from the Altera Corporation and HP. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the sponsors or any other entity.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **DEC 2003**	2. REPORT TYPE	3. DATES COVERED **00-00-2003 to 00-00-2003**

4. TITLE AND SUBTITLE **Spatial Computation**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **225**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

Keywords: spatial computation, optimizing compilers, internal representation, dataflow machines, asynchronous circuits, reconfigurable computing, power-efficient computation

Abstract

This thesis presents a compilation framework for translating ANSI C programs into hardware dataflow machines. The framework is embodied in the CASH compiler, a Compiler for Application-Specific Hardware. CASH generates asynchronous hardware circuits that directly implement the functionality of the source program, without using any interpretative structures. This style of computation is dubbed "Spatial Computation." CASH relies extensively on predication and speculation for building efficient hardware circuits.

The first part of this document describes Pegasus, the internal representation of CASH, and a series of novel program transformations performed by CASH. The most notable of these are a new optimal register-promotion algorithm and partial redundancy elimination for memory accesses based on predicate manipulation.

The second part of this document evaluates the performance of the generated circuits using simulation. Using media processing benchmarks, we show that for the domain of embedded computation, the circuits generated by CASH can sustain high levels of instruction level parallelism, due to the effective use of dataflow software pipelining. A comparison of Spatial Computation and superscalar processors highlights some of the weaknesses of our model of computation, such as the lack of branch prediction and register renaming. Low-level simulation however suggests that the energy efficiency of Application-Specific Hardware is three orders of magnitude better than superscalar processors, one order of magnitude better than low-power digital signal processors and asynchronous processors, and approaching custom hardware chips.

The results presented in this document can be applied in several domains: (1) most of the compiler optimizations are applicable to traditional compilers for high-level languages; (2) CASH itself can be used as a hardware synthesis tool for very fast system-on-a-chip prototyping directly from C sources; (3) the compilation framework we describe can be applied to the translation of imperative languages to dataflow machines; (4) we have extended the dataflow machine model to encompass predication, data-speculation and control-speculation; and (5) the tool-chain described and some specific optimizations, such as lenient execution and pipeline balancing, can be used for synthesis and optimization of asynchronous hardware.

Contents

Chapter 1

Introduction

1.1 Motivation

For most computer users the speed and capabilities of today's computer systems were unthinkable as little as a decade ago. The relentless advance of technology at an exponential pace for more than 40 years has produced amazingly complex and powerful machines – in September 2003 Intel released the Itanium 2 processor for servers, built out of 410 million transistors on a single 374mm^2 chip and consuming 130W. A desktop microprocessor with more than 1 billion transistors is expected by 2007. The exponential increase in computing resources is expected to last at least another decade, and perhaps even more if the promises of nanotechnology come true.

However, all this progress has come at a cost. The very success of miniaturization creates new unexpected obstacles. Figure 1.1 illustrates one such difficulty: the complexity of the hardware design increases with the number of transistors, at an exponential pace [itr99]. For instance, the complexity of a hardware design depends on the number of *exceptions* that cannot be automatically handled by Computer-Aided Design (CAD) tools [HMH01]; the number of these exceptions grows with the number of transistors. Although the productivity of hardware designers increases too, due to improvements in CAD tools, it does so at a slower pace. The end result is a growing *productivity gap* between what we have and what we can economically use. Fewer and fewer companies can close this gap, and when they do, they employ larger and larger design, test, verification and manufacturing teams.

The problems of hardware design productivity are nowhere near being solved. In fact, more ominous obstacles are surfacing. Figure 1.2 shows the signal propagation delay through logic gates and "average-sized" wires. Since many structures of a modern processor such as the forwarding paths on the pipeline, the multi-ported register files, the instruction wake-up logic, etc., require global signals (i.e., spanning multiple modules), there simply is not enough time for the signals to propagate along these wires at very high clock speeds. In fact, at 10GHz a signal can cover less than 1% of the surface of a chip in one clock cycle [AMKB00], making any architecture relying on global signals infeasible.

The research presented in this document is aimed directly at these problems. We explore **(A)** a new computational model which requires no global synchronization, and **(B)** a new CAD methodology aimed at bridging both software compilation and microarchitecture. Our model is named Spatial Computation. In Spatial Computation, applications written in high-level languages are compiled directly into hardware circuits. The synthesized circuits feature only localized communication, require no broadcast, no global control, and are timing-insensitive, and therefore correct even when the communication latency is not statically predictable. The compiler we have developed requires no designer intervention (i.e., it is fully automatic),

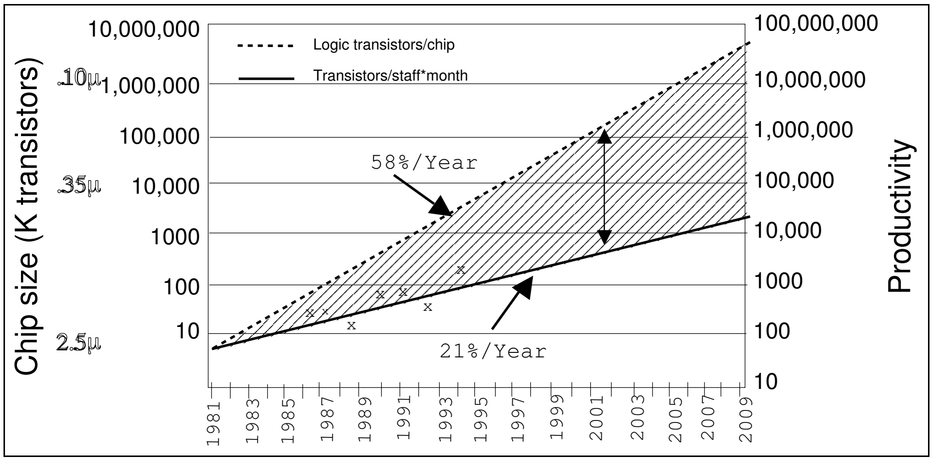


Figure 1.1: *Complexity and productivity versus technology generation.*

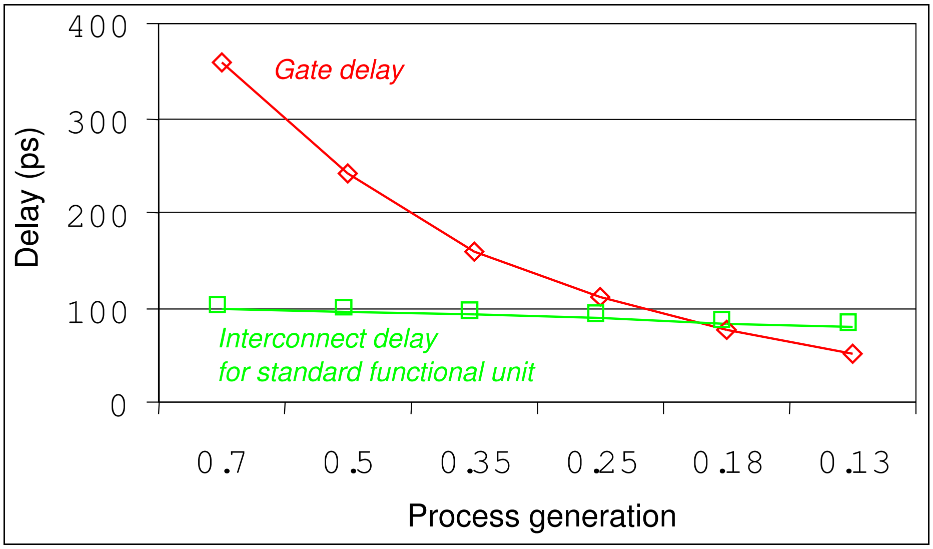


Figure 1.2: *Wires and not gates dominate the delay in advanced technologies.*

is fast, and exploits both instruction-level parallelism and pipelining. This thesis is an investigation of the compilation methodology and of the properties of the synthesized circuits.

1.2 Application-Specific Hardware and Spatial Computation

One can view Spatial Computation in several ways:

- As a pre-von Neumann model of computation [vN45]: a computer which has no stored program in the form of a sequence of instructions. In this sense, ASH is a big automatically-generated Application-Specific Integrated Circuit (ASIC).

- As a *specialization* of the pipeline of a processor through the "unrolling" and "constant folding" of a fixed program, as illustrated in Figure 1.3.

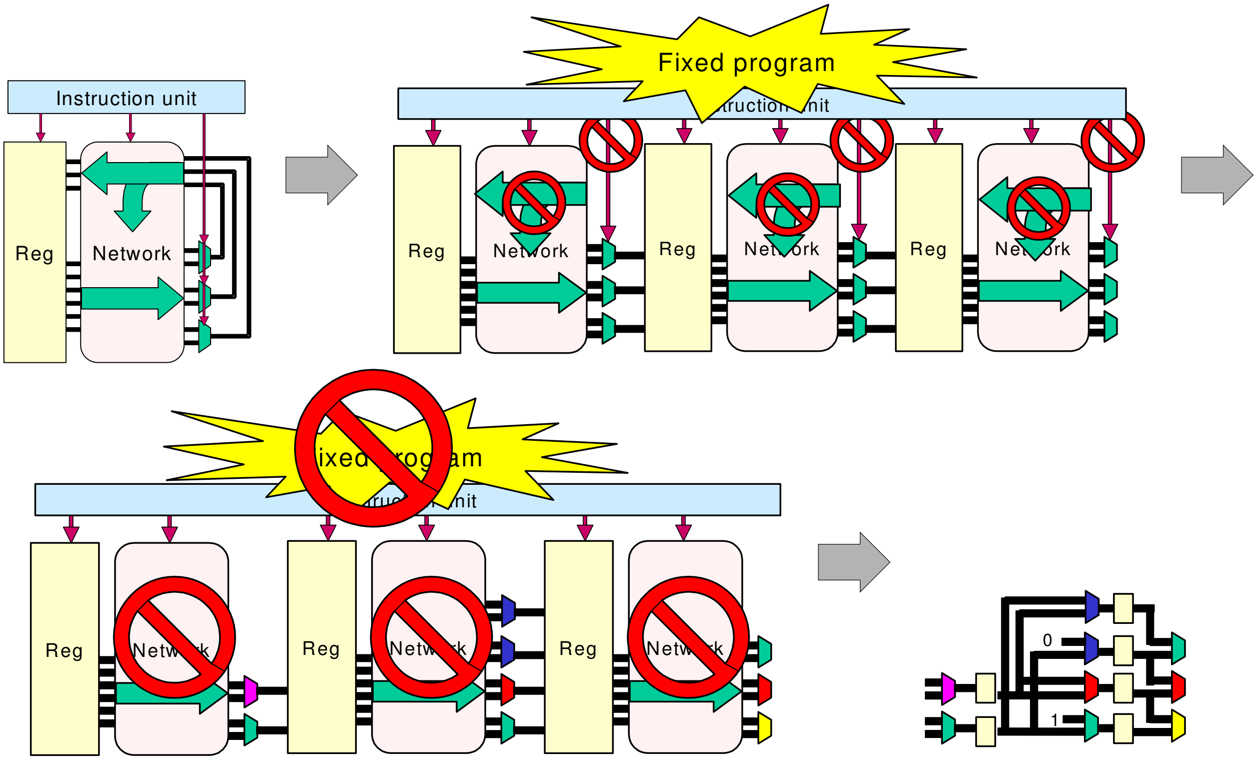


Figure 1.3: *ASH can be seen as the unrolling and specialization of the pipeline of a traditional wide processor, by removing the forwarding paths, specializing the type and number of functional units, simplifying the interconnection network to point-to-point links, removing the global register file and instruction issue logic.*

- As the limit result of clustering a very long instruction-word (VLIW) processor with a virtually unlimited instruction word.

From an architectural point of view Spatial Computation exhibits some desirable attributes:

- It is *program-specific* and therefor it has no interpretation overhead.
- It can exploit virtually unlimited instruction-level parallelism (ILP).
- It features mostly short and fast point-to-point wires driven by a single writer each (i.e., wires that require no arbitration).
- It is asynchronous and therefore latency tolerant. This makes it correct by design, even when it employs global structures, such as the memory access network.
- As in superscalar processors and dataflow machines, its computation is dynamically scheduled, based on the availability of data.
- It requires no centralized control structures.

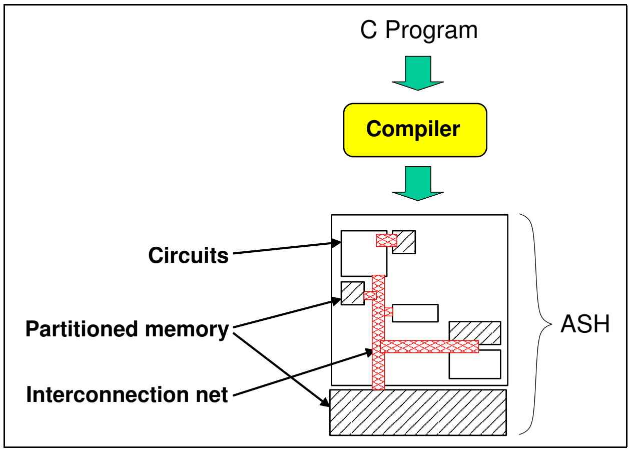


Figure 1.4: *Application-Specific Hardware is generated by the automatic synthesis of computational structures, memories and an interconnection network from C programs.*

- It has a modular structure.
- It is composed of *lean* hardware, in particular, both datapath and control-path contain no broadcast structures, no arbitration, no multi-ported memory or register files, no content-addressable structures, and would therefore be able to run at a very high speed.[1]
- It is easy to reason about. Therefore we believe it is easily amenable to formal verification.

In this thesis we investigate in detail a particular instance of Spatial Computation, with the following features:

- Each source procedure is translated into a distinct hardware structure.
- Each source operation is synthesized as a different hardware arithmetic unit. As a result, our circuits exhibit *no computational resource sharing*.
- All program data is stored in a single monolithic memory, accessible through a conventional memory hierarchy, including a load-store queue and caches.
- The synthesized hardware is an application-specific *static dataflow machine*. This means that (1) operations are executed based on the availability of data; and (2) at any point in time there can exist in the circuit at most one result for each operation.[2]

We define Application-Specific Hardware (ASH), as the hardware structure implementing a particular program, synthesized under compiler control from the application source code. The translation is illustrated in Figure 1.4.

From now on, when we use the terms “Spatial Computation” and “ASH” we will only refer to this particular architecture. There can be other reasonable definitions for these terms (e.g., application-specific processor), but here we use these terms with the meanings as defined.

[1] However, the memory access network requires some complex structures.

[2] In contrast, in dynamic dataflow [AN90], at some point in time, there can be an arbitrary number of results “live” for each operation.

1.2.1 ASH and Computer Architecture

The features of Spatial Computation suggest that it may provide very good performance on data-intensive programs, which have high amounts of instruction-level parallelism. Indeed, as we show in Chapter 7, ASH can surpass by a comfortable margin a 4-wide out-of-order superscalar on media processing kernels.[3]

However, as our analysis shows, some of the strengths of ASH are also its weaknesses when we consider control-intensive programs. The lack of branch prediction in ASH is a severe handicap for benchmarks where the ILP is low and the computation of the branch conditions is on the critical path. The distributed nature of ASH makes reaching memory expensive and squashing speculation difficult.

These behaviors suggest that a versatile computer system should combine the strengths of both out-of-order processors for control-intensive code, and Spatial Computation for data-intensive kernels. This research opens the exploration for the detailed architecture of such a system, and mostly of a low-overhead hardware substrate suitable for Spatial Computation (when compared to contemporary FPGA devices [ZRG$^+$02]).

1.3 Research Approach

A good characterization of this research would be "translating programs to hardware in the absence of resource constraints." This approach is certainly extreme and could be dismissed as relying on unrealistic assumptions. However, we think that there is merit in such a methodology:

1. First, such an approach forces a change of perspective in treating the problem and may provide fresh insights into the solution space, allowing one to reconsider methods traditionally excluded. As a result, even though the constraints are relaxed, such an approach may produce results usable within a realistic setting.

   For example, Spatial Computation blurs the notion of instruction set architecture (ISA). This enabled us to explore the addition of new computational primitives, such as the token generator (whose use is described in Section 4.5.4), which dynamically controls the slip between two loops. The use of this primitive enables a new type of parallelization/pipelining optimization, which we named loop decoupling. An important but subtle aspect of the lack of the ISA is the fact that the computational primitives no longer have to be encoded in fixed-size machine instructions. This enables us to use primitives with a potentially large number of inputs, such as multiplexors or token combine operators (the latter described in Section 4.4). We currently are investigating how the use of primitives similar to token generators in a more traditional multithreaded architecture can enable the same benefits.

   The register promotion algorithm from Section 4.6 is the result of a similar shift in the point of view regarding the "layering" of a system: that algorithm takes a traditional compile-time construct — dataflow availability information — and transforms it into a run-time object. This transformation enables an accurate dynamic computation of availability and therefore obtains dynamic optimality of the number of memory accesses. As the computational bandwidth of processors increases, these kinds of optimizations may become more advantageous. In the case of register promotion, the benefit from removing expensive memory operations outweighs the increase in scalar computations to maintain the dataflow information. We named this new kind of dataflow analysis — Static Instantiation, Dynamic Evaluation (SIDE). The compiler statically determines which dataflow facts are beneficial in a run-time computation, and inserts code in the program to dynamically compute the dataflow facts. The register promotion algorithm is the application of the SIDE method to the classical dataflow

[3] This is assuming our assumptions for the architectural parameters are realistic.

availability problem. We are currently investigating whether the SIDE methodology can be applied to other dataflow analyses.

Another example of the power of an unconventional framework is shown by the algorithms we have devised for treating memory dependences. In traditional systems, memory dependence information is computed first by the compiler and then completely discarded and re-created dynamically by the processor. So there can be some benefit if the static and dynamic stages can share information. In Spatial Computation the token edges used to ensure memory consistency are both a compile-time and a run-time object. We are currently investigating the benefit of such a scheme in classical architectures.

2. Second, while we relax some constraints, we do maintain some hard limits for other parameters. For example, we do not modify the semantics of C, we do not exclude any language constructs, and we do not enforce a particular coding style for programs translated to hardware, as does the vast majority of the research on high-level synthesis from C.

3. Third, our results provide understanding on the limits of a particular model of computation, as evidenced in Chapter 8, which compares ASH with superscalar processors.

1.4 Contributions

Even though the research presented in this document is only the beginning of a long-term endeavor, and many more refinements need be made before a complete end-to-end solution is obtained, it still has produced some worthwhile results. To our knowledge, the following points are original research contributions of this work:

- **Predicated Static-Single Assignment dataflow internal representation:** We are the first to report on the wide scale usage of a compiler internal program representation that brings together static single-assignment, predication, forward speculation and a functional representation of side-effects. Several features of this representation make compiler development particularly efficient: (1) it has precise semantics; (2) enables a succinct expression of most common program optimizations; (3) it enables efficient reasoning about memory state and therefore enables powerful memory optimizations; and (4) it makes dataflow analyses simple and efficient.

- **SIDE as a new framework for dataflow analysis:** Our new hybrid dataflow analysis framework, Static Instantiation Dynamic Evaluation, uses code to dynamically evaluate dataflow information when a static evaluation is too conservative.

- **New compiler algorithms:** We describe new optimization algorithms for redundancy elimination in memory accesses: register promotion and partial redundancy elimination for memory. We also describe algorithms for enhancing the pipelining of loops, such as a scheme for pipeline balancing, memory access pipelining in loops through fine-grained synchronization, and loop decoupling.

- **Language extension:** We have studied the efficiency of user annotations in C programs (through the use of `#pragma` statements) to convey pointer non-aliasing information to the compiler.

- **Dataflow model extensions:** We have realized the first translation of the complete C language to a dataflow machine. This effort has crystallized a methodology for implementing imperative languages, including unstructured flow-of-control, recursion, side effects and modifiable storage. We have incorporated predication and speculation in the dataflow model of execution and we have suggested a way to perform the equivalent of branch prediction in dataflow machines. Other contributions to the dataflow model of execution are the use of lenient operations and an enhanced form of pipeline balancing.

- **Hardware synthesis:** We have built the first complete system able to translate arbitrary C programs into hardware. This tool differs from most prior approaches in its change of perspective: C is not regarded as a hardware-description language, destined to describe an independently-executing piece of hardware. Instead, the goal of the synthesis system is to directly implement a given application as a hardware circuit.

- **Asynchronous circuit design:** We have provided the first tool that can be used to translate C programs into asynchronous circuits. The translation methodology is general enough to be applicable to any imperative language.

- **Embedded systems construction:** We have built the CASH compiler, which can be used as a tool for very fast prototyping of embedded systems hardware accelerators and has shown its effectiveness on a wide range of media processing applications. Although not detailed in this document, we have suggested a new way of performing hardware-software partitioning, which relies on separating the policy from the mechanism. This partitioning is easily amenable to automatization. CASH, together with the partitioning methodology, constitute a very efficient method of exploring the design of complex application-specific system-on-a-chip devices using only the application source code.

- **New perspective on superscalars:** We have performed a new limit study on the subject of instruction-level parallelism and the effectiveness of architectural features for exploiting it. Namely, by contrasting a static dataflow model with unbounded resources (ASH) against a superscalar processor implementation, we have highlighted the effectiveness of a monolithic architecture at performing memory accesses, prediction and speculation, and emulating a full dynamic dataflow model of execution.

1.4.1 Thesis Statement

Software compilation technology for targeting predicated architectures can be naturally adapted for performing the automatic synthesis of application-specific, custom hardware dataflow machines. This compilation methodology translates media processing kernels into hardware with a high degree of instruction-level and pipeline parallelism. However, the resulting distributed computation structures are not as easily amenable as traditional monolithic superscalar processors at using acceleration mechanisms such as (correlated) prediction, speculation and register renaming. The lack of these mechanisms is a severe handicap when executing control-intensive code.

1.5 Thesis Outline

This document has been divided into two parts. Part I, encompassing chapters 2–5, is concerned with the “static” aspects of ASH, i.e., compilation. Chapter 2 is a broad overview of the compilation process. Chapter 3 describes in detail Pegasus, the internal representation of the CASH compiler and the process of building it starting from a control-flow graph. Chapter 4, the most substantial of this thesis, is devoted to a description of the program transformations performed on Pegasus with the goal of increasing performance. Chapter 5 is a collection of advice for other developers of research compilers.

Part II of this thesis, encompassing chapters 6–8, is about the dynamic behavior of ASH. Chapter 6 looks at some emergent properties of the execution of ASH fabrics and discusses some particular optimizations that are only applicable for such a model of computation. A particularly important subject covered is dataflow software pipelining, the main source of parallelism exploited in Spatial Computation. The next two chapters are all about performance. Chapter 7 uses ASH to implement kernels for embedded-type benchmarks and compares its speed with both VLIW and superscalar processors. It also evaluates the performance of the generated circuits using low-level Verilog simulations. Chapter 8 dwells on the hypothetical capabilities of ASH for executing complete control-intensive programs, and contrasts those capabilities with the ones available in a superscalar processor.

Appendix A describes briefly our benchmarking methodology. Appendix B contains both some correctness arguments about the behavior of ASH and a formal definition of the semantics of all its computational primitives.

Part I

Compiling for Application-Specific Hardware

Chapter 2

CASH: A Compiler for Application-Specific Hardware

This chapter is devoted to a bird's eye view of CASH, the compiler translating C programs into Application-Specific Hardware (ASH). While the subsequent chapters go into many more details, the material here justifies some of the design decisions and describes the compilation process end-to-end.

2.1 Overview

Figure 2.1 shows CASH within its environment. The inputs to the compiler are programs written in ANSI C.[1] CASH represents the input program using a dataflow internal representation called Pegasus. The output of CASH is a custom hardware dataflow machine which directly executes the input program. The result can be implemented either as a custom hardware device or on a reconfigurable hardware substrate.

We envision two different uses for CASH:

- CASH can be used as a pure CAD tool, using C as a hardware description language (HDL) and generating a complete hardware implementation of a program. An evaluation of CASH from this perspective is given in Chapter 8.
- CASH can be used as a piece of a more complex tool-chain targeting a hybrid system-on-a-chip. CASH can then be used to translate selected parts of a larger C program into hardware. The remaining code is executed on a traditional microprocessor, relatively tightly coupled with the custom hardware. Chapter 7 analyses the effectiveness of this configuration.

CASH translates the input program into a flat computational structure which contains no interpretation engine. Roughly, to each operation in the source corresponds one arithmetic unit in the output circuit. This kind of computation structure is called Spatial Computation. It exhibits maximum parallelism and minimal resource sharing. The only resource sharing is at the procedure level since different calls to the same procedure re-use the same circuit.

Currently CASH translates each procedure of a program into a different, completely independent computational circuit. However, by using inlining and by writing carefully crafted library functions[2] one can control to some degree the sharing of the structures in the resulting circuit.

[1] The Kernighan & Ritchie dialect mostly works, but may cause occasional difficulties.

[2] For example, by implementing all floating-point operations as library calls.

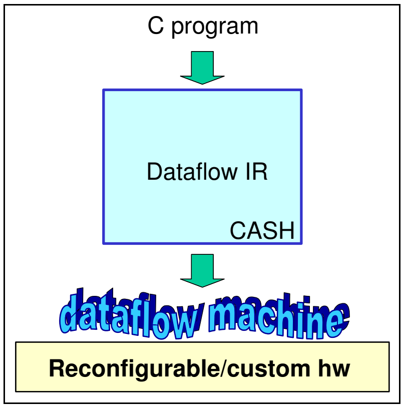


Figure 2.1: *The CASH compiler translates C programs into hardware.*

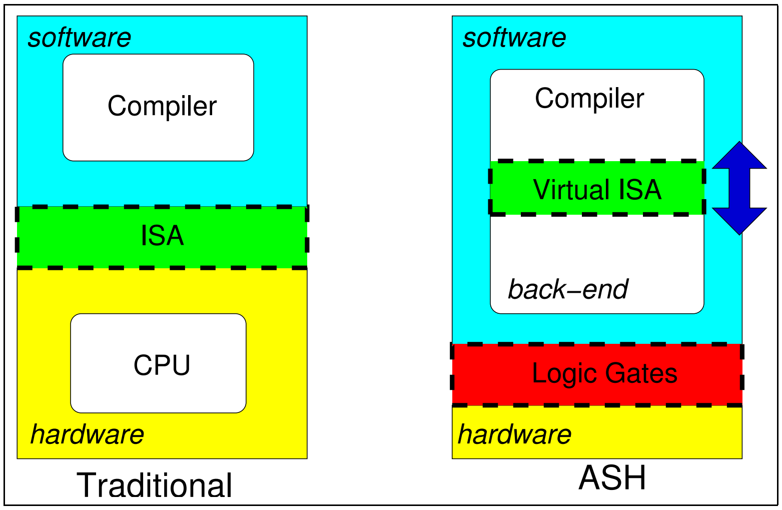


Figure 2.2: *The interface between hardware and software in traditional settings and in ASH.*

2.2 The Virtual Instruction Set Architecture

Figure 2.2 shows the relationship between hardware and software in two settings: the traditional processor-centric world to the left and ASH to the right. In the traditional setting the software and hardware worlds are separated clearly by the Instruction Set Architecture (ISA), which is a tight contract between the two worlds. The ISA is basically the set of machine instructions recognized by the CPU. The compiler has to express any program as a combination of these instructions. The traditional ISAs are quite rigid, producing a “tyranny of the ISA”: even if the compiler has more knowledge about the application at hand, it has no way to send this information to the hardware. Rebellion against this tyranny is quite difficult, as illustrated by the case of the Itanium microprocessors. These processors were designed to accommodate advances in compiler technologies, and therefore required radically different ISAs compared to the traditional x86 processors manufactured by the same company. Intel has needed more than 20 years to decide to switch to this new ISA, despite known shortcomings in the x86 processors. Mistakes in the design of an ISA are very hard to fix.

This state of affairs changes completely in ASH, as shown to the right in Figure 2.2. The interface

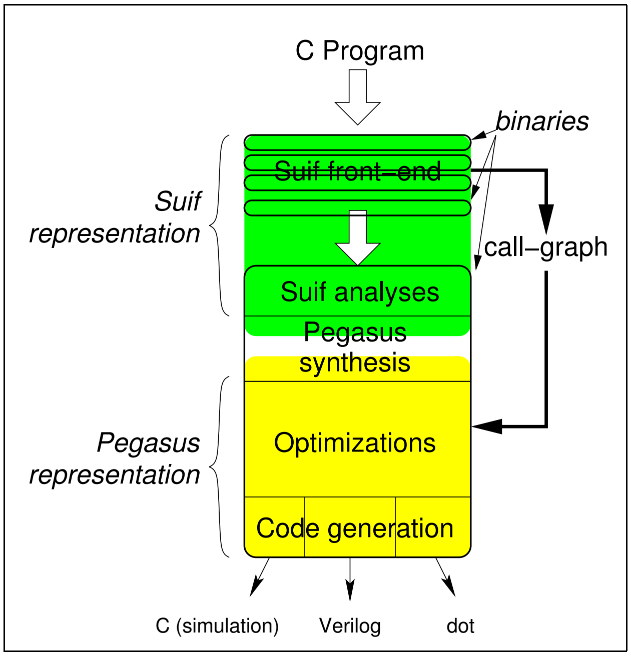


Figure 2.3: *The CASH compiler is composed of a Suif front-end for carrying out parsing and high-level optimizations and a custom back-end using a new internal representation called Pegasus.*

exposed by the hardware is composed of raw computation gates. This frees, but also complicates, the compilation process. In order to handle the great semantic gap between the source code and the target machine our approach consists in building a "virtual ISA." This time the ISA is entirely a compile-time notion. Think of the virtual ISA as the set of computational primitives the compiler uses to express the meaning of the program. In our setting the virtual ISA is the interface between the compiler core and the back-ends. Unlike in the traditional setting, the virtual ISA is not bound by any rules and can be changed. CASH's virtual ISA is the set of computation primitives of the Pegasus intermediate representation, described in Section 3.2. While virtual ISA resembles a traditional ISA in many respects, it also features substantial differences. For example, it is not bound by machine resources, i.e., does not require a fixed encoding (say, 32 bits) for each instruction. Second, it may be changed by the compiler implementer without touching the external hardware/software interface. The Pegasus virtual ISA contains constructs usually not found in traditional processors, such as a direct representation of memory dependences, described in Section 3.2.3.5. Exploring the space of virtual ISAs may be a rich and rewarding research activity, which may also suggest improvements to traditional ISAs.

2.3 Compiler Structure

The CASH compiler is a tool-chain consisting of a Suif-based [WFW$^+$94] front-end, a custom-developed core, and several back-ends tightly integrated with the core, all tied together by Perl scripts and Makefiles, as shown in Figure 2.3. Occasionally we will refer to the core also as CASH.

CASH is built on top of the Suif 1 infrastructure — the code base is Suif 1.3.0.5. No optimizations are used from the original Suif distribution. The only functionality used consists in parsing and un-parsing, linking, and basic object manipulation of symbol tables, instructions and static data initializers. All major

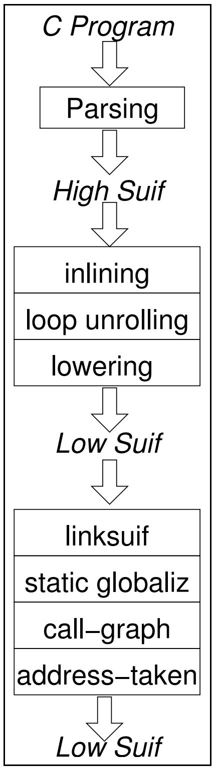


Figure 2.4: *Suif front-end structure. The passes communicate with each other through Suif files.*

Suif optimizations and analyses used in CASH are locally developed (however, as described below, some are borrowed from different research groups).

2.3.1 Front-End Transformations

The front-end is composed of multiple Suif compilation passes built as separate executables communicating through Suif files. Figure 2.4 shows the transformations performed in the front-end. The front-end will be rarely mentioned in this document.

- Parsing is performed by the Suif driver `scc`. It is invoked without any optimizations. The result of parsing are high-Suif representation files (i.e., containing high-level constructs, such as `for`, `while` loops and `if` statements, nested scopes and array accesses).

- The procedure inlining pass was written by Tim Callahan from Berkeley as part of his GarpCC [CW98] compiler. It is highly configurable, allowing the use of various policies for inlining. Currently we do not perform cross-file inlining, although the pass supports this functionality.

- The loop unrolling was written by Todd Mowry, then at Stanford. It can unroll both `while` and `for` loops, both with and without compile-time constant bounds. Loop unrolling has been tuned to aggressively unroll loops, proving very beneficial for embedded-type benchmarks (see Chapter 7).

Currently we do not unroll loops containing function calls[3] or expensive operations, such as division.[4]

- The lowering pass removes all high-level constructs and nested scopes; the result of lowering is low-Suif, roughly a three-operand representation of the original program.

- Linksuif is a Suif pass which unifies the symbol tables of several files allowing them to be used as a unit. It is a prerequisite of the following passes.

- The "static globalization" pass transforms procedure `static` variables into globals by mangling their names and moving them to the global symbol table. It is solely used to remove the syntactic sugar of `static` locals, which are just globals with a local visibility.

- The call-graph pass computes a conservative approximation of the program call-graph. The main usage of the call-graph is to detect recursive procedures, which require a more heavyweight approach in CASH. Since call-graph computation for a language with pointers is P-SPACE complete [Wei80], we resort to several approximations. First, the whole program is scanned and all procedures whose address is taken are noted. Second, a flow-sensitive, intra-procedural pointer analysis is used to restrict as much as possible the points-to set of all functions called through pointers. Third, the strong type-based heuristics described in [Atk02] are applied to prune the candidates for each call through pointers. The strongly-connected components are computed — these contain exactly the recursive functions. The resulting call-graph is written to a text file and cached (i.e., it is not recomputed unless the source program changes). It is important to note that it is not enough to know which procedures are recursive. It is also necessary to know which call instructions are potentially recursive.

- Finally, the "address-taken" pass is a Suif built-in flow-insensitive analysis, which tags all objects whose address is taken (and global variables as well). Next, all accesses to such objects are transformed to explicit memory loads and stores through pointers. This step clearly separates scalar accesses from ambiguous accesses which can have side-effects.

2.3.2 Core Analyses and Transformations

The core of the compiler is the main subject of this thesis. It consists of two parts: a translation from the Suif intermediate representation to our own predicated representation called Pegasus; and the bulk of the program transformations and optimizations, carried on Pegasus. Unlike the front-end, the entire processing of the core and the back-ends are tightly integrated in a single binary executable. Figure 2.5 illustrates the structure of the core.

In order to simplify the build process of the Pegasus representation several analyses are carried out on the three-operand code. Let us briefly discuss them, as shown in the top part of Figure 2.5. The complete implementation of the core was developed by us. It uses some Suif libraries that we have developed to build control-flow graphs and to perform generic dataflow analyses.

- If the call-graph is supplied, it is read at this moment. The call-graph structure is used both to construct the control-flow graph (CFG) and later in the compilation process to synthesize code differently in recursive/non-recursive procedures.

[3] There are two reasons for not unrolling loops with function calls: (1) the performance of the unrolled loop cannot be predicted statically; and (2) it changes the pre-computed call-graph of the program.

[4] The reason for not unrolling such loops is detailed in Section 6.8.

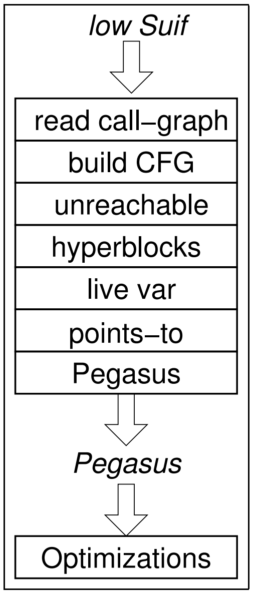


Figure 2.5: *The structure of the core of the compiler.*

- Building the CFG is fairly straightforward. However, as described in Section 3.3, some constructs are handled specially; namely, `return` instructions are always in a basic block of their own, and recursive procedure calls always start a new basic block.

- A simple reachability computation performs unreachable code removal on the CFG.

- The CFG is partitioned into disjoint hyperblocks. This algorithm is based on a depth-first traversal of the CFG and is discussed in Section 3.3.3.

- A live variable computation, which marks for each program point all live scalar variables. There is no lifetime analysis of memory-resident objects, only of the virtual registers.

- A simple pointer analysis is carried out. This step computes a read-write set for each memory access instruction (load, store, call). The set contains symbolic memory locations that may be accessed at run-time by the instruction. For procedure calls it is conservatively assumed that all objects in the read-write set may be modified within the callee and its children.[5] The read-write set is used later during the Pegasus build to insert token edges between non-commuting memory operations (Section 3.3.9).

  Since the cost of a points-to analysis can be quite substantial (the space-complexity is $|CFG| \times |Vars|$, (i.e., the number of basic blocks multiplied by the number of symbolic objects that can appear in the points-to-sets), a simple heuristic may decide to forgo completely the points-to analysis if the CFG is too large. If this is the case, a very conservative approximation is used instead — all memory operations may access all memory locations. Just a handful of procedures in our benchmarks are too large for pointer analysis.

[5] The write-set of a call may be further reduced by using information about arguments qualified with `const`. But we have not implemented this feature yet.

The actual Pegasus building algorithm is described in detail in Section 3.3, while the Pegasus-based optimizations are covered in detail in Chapter 4. The order of the Pegasus optimizations is the subject of Section 4.2.

2.3.3 Back-Ends

Currently CASH has three back-ends, illustrated in Figure 2.3:

- A dot back-end used to generate drawings of the internal representation of one compiled function. The representation is a fairly simple textual representation of the Pegasus graph in the language of the Graphviz [GN99] graph visualization tool. This “back-end” can be invoked also from a debugger, to print the state of the representation at various points during the compilation process. The `dot` output has a very simple structure and is easily manipulated by various Perl scripts for debugging and illustrations. The `dot` tool automatically computes the graph layouts from the textual description. Many of the illustrations in this text were produced in this way. The conventions used for drawing Pegasus representations and some shortcomings of `dot` are discussed in Section 3.2. The use of this back-end for debugging and performance analysis is further described in Section 5.3, Section 6.3 and Section 6.6.

- A C-simulator back-end is used to generate a relatively high-level simulation of each program. This simulator was used to gather most of the numeric results presented in this document. The simulator models the finite-state machine of each asynchronous operation and is parameterized with operation latencies. However, the simulator ignores the effects caused by the physical layout.

- A Verilog back-end generates direct asynchronous circuits from the Pegasus representation. This back-end was written entirely by Girish Venkataramani.

  Most of the translation is fairly straightforward, using libraries for technology mapping each Pegasus primitive. The generation of the memory access network is somewhat complex; the network for handling procedure calls is still under design. The discussion of this back-end is out of the scope of this document. Preliminary measurements using the Verilog code generated by this back-end are presented in Section 7.5. These simulations are more accurate than the ones of the C simulator and do include the effects of the technology mapping libraries and circuit layout.

2.4 Optimizations Overview

The rest of this document is only concerned with the core of CASH compiler, as described in Section 2.3.2, operating on the Pegasus representation. In this section we briefly overview the optimizations performed by CASH on the compiled program. A detailed discussion of this topic is the subject of Chapter 4.

2.4.1 What CASH Does Not Do

Before we summarize the optimizations performed, let us mention some optimizations which are *not* part of our compiler.

- A peculiarity of the Pegasus representation is that it is a pure dataflow representation (see Chapter 3). As such, it does not contain any control-flow information.[6] In consequence, many control-flow related optimizations are irrelevant (but see Section 4.1 for a discussion).

- Since CASH targets Spatial Computation, which does not have a fixed instruction set, its back-ends are substantially different from traditional compilers. For instance, CASH does not contain either a register allocator (because we can allocate arbitrarily many registers) or a scheduler (because the circuits synthesized are asynchronous, dynamically scheduled).[7] It also does not contain the lowest level optimizations, such as instruction selection (because we can implement almost any instruction), peephole optimization, or instruction-cache optimizations (because there is no instruction issue in Spatial Computation).

- Although CASH is inspired in many respects from the compilation methodology for Explicitly Parallel Instruction Computing (EPIC) architectures [ACM$^+$98], it does not use at all profiling information for guiding its decisions. It is a "pure static" compiler. While the use of profile information would certainly help improve some of the code transformations, it requires a more sophisticated tool-set for mapping back run-time information to the original source-code. We believe that our static approach still provides an interesting reference point in the design space. Some of the features of Spatial Computation, such as lenient execution (see Section 6.4), can make up for the lack of profiling information. Investigating the use of run-time hints for improving Pegasus is an interesting subject of future research.

- CASH is a C compiler. As such it does not implement optimizations which are useless for C programs, such as array bounds checking removal or exception handling. The latter is actually a liberating assumption — since C does not really have an exception model, CASH does not implement any (but see Section 2.5.3.1 for a discussion).

- CASH is not a parallelizing compiler. As such, it does not perform any of the loop- and data-restructuring transformations customarily found in parallelizing FORTRAN compilers. The C language is not easily amenable to such transformations.

- CASH does not do any inter-procedural optimizations, except a special handling of recursion and a rather traditional procedure inlining. This is not really a limitation of the model. There is no fundamental reason that inter-procedural analyses and optimizations could not be added to a spatial compiler. This reflects our belief that scalable compilation methodology has to be modular and to support separate compilation. We think that the need for interprocedural analysis and optimization actually reflects a source language shortcoming. In Section 7.3 we explore the way a simple language extension (through the use of a C `#pragma` statement) can be used to supplant expensive inter-procedural aliasing information.

- While CASH tries to exploit parallelism at the program level, it still does too little in exploiting it at the memory level. We expect that reorganizing data layout can provide substantial benefits [SSM01, BRM$^+$99]. However, the C language makes such optimizations difficult, since memory layout has

[6] However, a compiler targeting a traditional machine could, with little difficulty, reconstruct a control-flow representation from Pegasus. Predication is deeply enmeshed in Pegasus and therefore targeting an architecture without direct support for predication would probably be rather suboptimal.

[7] However, we do plan to add some scheduling-related optimizations, i.e., optimizations taking into account the expected execution order of the operations.

to be exactly preserved for all objects for which some program accesses cannot be disambiguated at compile-time.

- Finally, CASH could do much more in terms of hardware-targeted optimizations. The BitValue analysis [BSWG00], which discovers useless bits in C programs and can narrow the size of the words computed on, is not yet integrated into CASH. Many important hardware-related optimizations are relegated to the post-processing tools, (i.e., the hardware synthesis tool-chain used to process the Verilog output from CASH). CASH is also not taking full advantage of the immense freedom afforded by the Virtual ISA by using custom operations (except in some modest ways, such as using tokens at run-time (Section 6.5.2) and the token generator operation (Section 4.5.4)). In the simulations presented in this text, CASH ends up by using a rather restricted set of computational primitives, rather similar to the machine code of a contemporary processor (with somewhat fewer limitations on the instruction size, such as the register set or immediate values encoding). Potentially important performance improvements may be unleashed by the use of a truly arbitrary set of computational primitives, as afforded by a truly custom hardware target, as shown in the research about reconfigurable computing.

2.4.2 What CASH Does

With regard to the range of program transformations performed, CASH is much closer to a software compiler than to a CAD tool. We have implemented in CASH most textbook scalar optimizations. In addition, CASH has proven to be a great vehicle for implementing some memory-related optimizations, such as redundancy elimination and register promotion.

Most optimizations in CASH are applied at the level of hyperblocks, which tend to be relatively coarse program regions (see Section 3.3.3). Optimizations are therefore, as effectiveness goes, somewhere between purely local, basic-block-based optimizations, and global, whole-procedure optimizations. While it is very hard to quantify the quality of optimizations in a precise way, we believe that the quality of the code is closer to that achieved by global optimizations than to purely local transformations. An important point is that most innermost loops[8] are hyperblocks and therefore are treated as an optimization unit.

As already discussed, the front-end performs procedure inlining and loop unrolling. The CASH core performs (truly) global constant propagation, constant folding, partial redundancy elimination for both scalars and memory, a wide range of algebraic optimizations, loop-invariant code motion, scalar promotion based on powerful memory disambiguation, strength reduction, unreachable- and dead code removal and Boolean simplifications. The compiler also performs pipeline balancing for enhancing the throughput of the circuits which exhibit dataflow software pipelining.

2.5 Compiling C to Hardware

One can find many arguments that C is a strong contender for the title of “worst possible hardware description language.” The main argument about using C is its huge installed code base and its undisputed use in systems programming at all levels.

2.5.1 Using C for Spatial Computation

The question arises naturally: if we consider C a “difficult” language for hardware description, why use it in the first place? Our motivations are several:

[8] Unless they contain a recursive call, as described in Section 3.3.11.

- The C language has a huge installed code base. This makes CASH a domain-independent compiler, equally applicable for digital signal processing, scientific computation or almost any other domain.

- C is frequently the language of choice for describing reference implementations. Indeed, many complex standards now contain not only textual specifications, but also a sample implementation and criteria for correctness, such as bit-exactness. For example, the General Systems Mobile (GSM) standard for mobile telephony relies on an EFR speech codec defined by the European Telecommunications Standards Institute (ETSI) specifications, which is specified by a reference C program along with a set of testing sequences [Ins]. So CASH can be used as a very fast prototyping tool for implementing custom hardware starting from a specification — just feed the source program to CASH and you can obtain a hardware circuit in a matter of man-minutes.

- While some aspects of the compilation process in CASH are specific to C, most of the techniques are equally applicable to other (imperative) programming languages. In fact, the input of the core CASH compiler can be considered the Low Suif representation (see Figure 2.5), which is closer to machine code than to C. Any language that can be translated to three-operand code could therefore be a source for our spatial compiler. Moreover, the freedom that C allows in pointer manipulation makes the translation task harder. Therefore, if CASH works for C, it ought to be easier to extend to other languages. For example, handling FORTRAN would mostly require replacing the algorithm for memory dependence estimation. It is somewhat less obvious whether the translation and optimization methodologies employed in CASH can be successfully applied to an explicitly parallel language, which would arguably be a better source for hardware description.

Some other motivations for using C have more to do with the pragmatics of the research process than with the economics of the solution:

- Choosing such a small and well-understood language, and leveraging existing tools (e.g., Suif) for tedious chores, allows us to focus our effort on the translation process and not on language design or parser implementation.

- Our own expertise is mainly in the area of software compilation, and so we have chosen this approach. While the trade-offs are certainly different when creating hardware and software artifacts, we believe that a hardware-design tool-chain looking more, and acting as fast as a software tool-chain, would bring increased productivity benefits to hardware design. This is already one of the main critical resources in the design cycle of new devices.

- As described in the introduction, the original motivation of this research is to address the challenge of creating multi-billion device designs by effectively handling complexity. We have therefore generously given ourselves virtually unlimited hardware resources. By constraining the research to the use of C, and making a requirement to handle all language features, we have anchored one end-point of the design space in “reality.” This allows us to evaluate the effectiveness of our methodology much better than if we were using a custom designed, special-purpose language.

2.5.2 Related Work

While there is a substantial amount of research on hardware synthesis from high-level languages, and even from dialects of C and C++, none of it supports C as fully as CASH does. One major difference between our approach and the vast majority of the other synthesis from C efforts is the difference in the role of the language itself. Most projects use C as a hardware description language (HDL), used to describe the functionality of a particular piece of hardware. We are not interested in using C as a description language. We are just interested in *implementing* the C program so that it produces the expected result.

Another way to express the difference between using C as an HDL and our approach is with regard to *reactivity*: hardware designs customarily assume that the environment can change at any moment and that the hardware must react immediately. In contrast, our view of the environment is extremely restricted. Even when CASH compiles only part of a program, it implicitly assumes that there is a single thread of control and that there are no unexpected external changes.[9] The stack (last-in, first-out) nature of procedure invocation/completion is also a typical aspect of software assumed by CASH.

This difference of optics is translated in a different approach. Other projects either add constructs (such as reactivity, concurrency and variable bit-width) to C in order to make it more suitable to expressing hardware properties, or/and remove constructs (such as pointers, dynamic allocation and recursion) which do not correspond to natural hardware objects, obtaining "Register Transfer C." Other efforts impose a strict coding discipline in C in order to obtain a synthesizable program. There are numerous commercial products using variants of C as an HDL [Mic99]: Cynlib from Cynapps [RR00, RR99], Cyber from NEC [Wak99, WO00], A | RT builder, now from Xilinx [Joh99, Joh00], Scenic/CoCentrinc from Synopsis [LTG97, Syn], N2C from CoWare [CoW00], compilers for Bach-C (originally based on Occam) from Sharp [KNY$^+$99], OCAPI from IMEC [SVR$^+$98], Compilogic's (acquired by Synopsis), System Compiler from C-level Design, (acquired by Synopsis) [lD99], c2verilog [SP98], Celoxica's Handel-C compiler [Cor03], and Cadence's ECL (based on C and Esterel) compiler [LS99]. University research projects include SpC from Stanford [SSM01], PACT from Northwestern [JBP$^+$02] and also [GKL99, Arn99, GSD$^+$02, GSK$^+$01, KC98]. For an interesting appraisal of the effectiveness of many of these tools, see [Coo00].

A completely different approach to hardware design is taken in the research on reconfigurable computing, which relies on automatic compilation of C or other high-level languages to target reconfigurable hardware [Wir98]. Notable projects are: PRISM/PRISM II [AS93, WAL$^+$93b], PRISC [RS94], DISC [WH95], NAPA [GM95], DEFACTO [DHP$^+$01], Chimaera [YSB00, YMHB00], OneChip [Esp00, WC96], RaPiD [ECF$^+$97], PamDC [TS99], StreamC [GSAK00], WASMII [TSI$^+$99], XCC/PACT [CW02], the systems described in [SSC01] and [MH00], compilation of Term-Rewriting Systems [HA00], or synthesis of dataflow graphs [RCP$^+$01]. Our approach is original because (1) it compiles arbitrary ANSI C programs, without restrictions or extensions; (2) it generates asynchronous, dynamically scheduled circuits; and (3) it generates circuits which implement directly the program, without the use of interpretive structures.

More remotely related is the work on "custom processors", whose goal is to synthesize a domain-specific processor. This is largely a problem of instruction-set design. In this group we can rank the Hewlett-Packard PICO project [SA02, MRS$^+$01, SGR$^+$01], the Tensilica products [WKMR01, Ten], and automatic "super-instruction" selection: [AC01, LCD02, ZWM$^+$01, CTM02, SRV98, WKMR01, LVL03], and many others. PICO handles only a restricted C subset, namely using dense matrix operations. Tensilica processors can be extended with arbitrary instructions, whose behavior is described in a Verilog-like language. These new instructions get automatically incorporated in the hardware and compilation tools. These methods often involve profiling execution traces of existing microprocessors and collapsing repeated patterns in custom instructions.

CASH has been profoundly influenced by the work of Callahan on the GARP C compiler [Cal02, CW98], whose ideas were incorporated in the Nimble compiler [LCD$^+$00]. GarpCC is still one of the

[9] Of course, some of the standard C language mechanisms, such as `volatile` qualifiers, can be used in the same way as in system code to prevent optimizations from relying too much on the model of single thread of control when external effects need to be modeled.

systems with the broadest scope targeting C to a reconfigurable fabric.

Our goals are most closely related to the "chip-in-a-day" project from Berkeley [DZC$^+$02], but our approach is very different: that project starts from parallel statechart descriptions and employs sophisticated hard macros to synthesize synchronous designs with automatic clock gating. In contrast we start from C, use a small library of standard cells, and build asynchronous circuits.

ASH circuits are somewhat more remotely related to other exotic computational models, such as WaveScalar [SMO03], the grid processor TRIPS [SNBK01], RAW [LBF$^+$98], SmartMemories [MPJ$^+$00], Imagine [RDK$^+$98], and PipeRench [GSM$^+$99].

2.5.3 Limitations

The current incarnation of CASH should be able to handle almost any portable legal C program, including pointers, structures, unions, and recursive functions. However, the following C constructs are not yet supported:

- Library functions. We do not compile the library functions, although they may account for a substantial part of the program execution time. ASH has no notion of I/O or system calls, so the library functions handling these could not be mapped to ASH anyway. In simulations we assume that the time spent in library functions is zero, both for ASH and for the baseline processors we consider.

- `alloca()`, the stack-based memory allocator. There is no fundamental reason `alloca` cannot be handled and support for it should appear in CASH soon.

- Functions with variable arguments (varargs). Again, there is no fundamental reason CASH cannot support varargs. Implementing varargs however will require a different parameter-passing implementation since CASH currently does never pass arguments on the stack.

- Exceptions, `signal()` and `exit()`: see below for a discussion of these interrelated subjects.

- `setjmp` and `longjmp` are not handled at all. `longjmp` could be handled by using an exception-handling mechanism to unwind the stack; `setjmp` would probably require the same kind of machinery as for saving the complete state of a recursive procedure, described in Section 3.3.11.

2.5.3.1 Exception Handling

Strictly speaking, the C language does not have any exception model. The semantics of a program that causes exceptions is left undefined by the C standard. CASH guarantees to never introduce exceptions which did not exist in the original program. If the source program executes safely, the compiled program should also do so. However, as many other C optimizing compilers do, CASH may remove some exceptions. For example, the register promotion algorithm may store variables in registers rather than memory, causing the optimized program to perform only a subset of the memory accesses of the source program. Currently we assume that run-time exceptions terminate the program.

Handling asynchronous function invocations (as does `signal()` in C) in Spatial Computation seems to be a very difficult task. In C the whole infrastructure for signal delivery and handling is really hidden within a substantial run-time support piece of machinery, usually built within the operating system kernel.

Exceptions are however important and need to addressed; at the very least, an exception should cause a clean program shutdown. It is not entirely obvious how shutdown (or reset) is to be achieved in the distributed computation structure of Spatial Computation.

A possible solution is to explicitly encode exceptions as control-flow within the program before proceeding to the program transformation and optimization. Exception handling done this way is likely to have a major impact on the efficiency of the generated code and may inhibit or complicate the task of some optimizations because potentially almost any memory access operation can generate an exception. A careful implementation would probably use some sort of "batch processing" of exceptions, accumulating exceptions, checking relatively infrequently for exceptions[10] and following a special execution path on exception detection. Function return values would need to be augmented to indicate the occurrence of exceptions in callees, to allow them to propagate through call chains.

At the very least, exception support is required for implementing the seemingly banal `exit()` function. As with `signal()`, the machinery behind `exit()` in a processor-based system is very complicated, involving the process destruction and resource reclaiming. A natural solution is to handle `exit()` as a synchronous, program-generated fatal exception and implement it in the way described above.

2.6 Code Complexity

By all measures, the core CASH is a small compiler. Table 2.1 shows the amount of source code used for the various parts of CASH, including the front-end, the back-ends and the libraries, as counted by the sloccount [Whe01] utility which ignores white spaces and comments. The line count for the libraries is only approximate since CASH does not always use the full functionality provided by each library. The first line of this table refers to the compiler components described in this thesis. The "build and optimize core" includes all the code described in Section 3.3, used to translate from a CFG representation to Pegasus, and all optimizations described in Chapter 4. A further breakdown of the code complexity for each optimization is presented in Table 4.1 in Section 4.1. The "build and optimize" core is the result of about 2.5 man-years of effort.

2.7 Compilation Speed

In this section we compare the compilation speed of CASH and `gcc` 2.95.1. The procedure is as follows:

- We disable inlining and loop unrolling in both compilers to control the code size, which has a direct bearing on the compilation speed.

- We compile with maximum set of optimizations enabled with both compilers. We use `gcc -O2`, since according to the `gcc` manual, it involves all optimizations except unrolling and inlining.

- For CASH we do not time the Suif front-end since it performs no useful operations in this configuration (except parsing). Since the front-end passes communicate through files, timing them would introduce too much noise.

- We stop compilation at the assembly level for `gcc`.

- We measure the complete compilation process end-to-end. Both compilers are fed all source files on a single command-line.

[10] For example, at the end of each hyperblock.

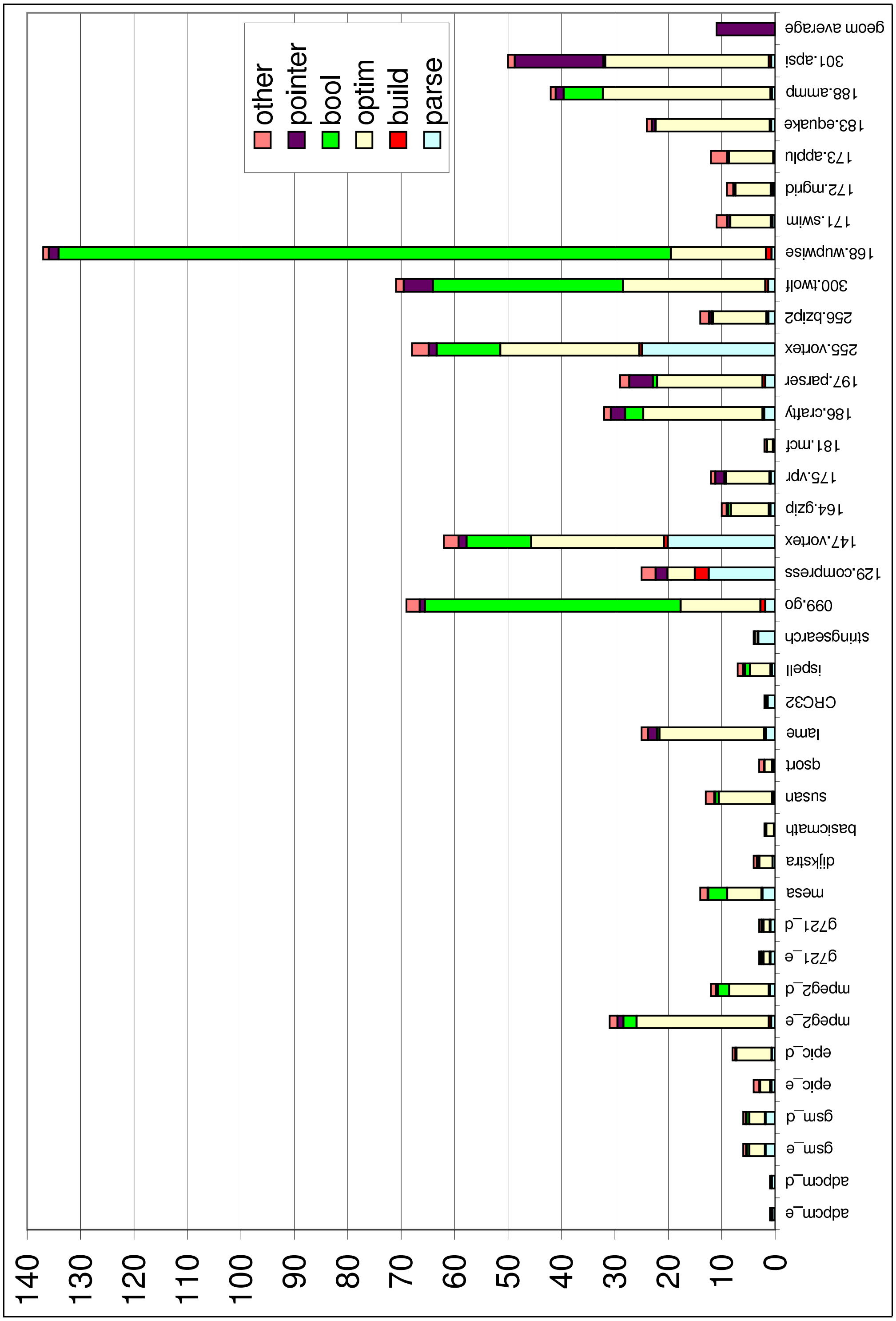


Figure 2.6: *Compilation speed slowdown compared to* `gcc` *(how many times CASH is slower). The time is broken down into: parsing the Suif intermediate representation, building the Pegasus representation, optimizations except Boolean simplification, Boolean simplification, intra-procedural flow-sensitive pointer analysis and other time (i.e., file I/O, code generation).*

Part	Functionality	LOC	Imported
Core	Build and optimize	15,517	
	gcc code	768	Y
	Verilog back-end	9.800	
	dot back-end	444	
	C back-end (based on Suif scc)	9,416	Y
Libraries	data structures and CFG and dataflow analyses	14,815	
	Espresso Boolean minimization	11,457	Y
	Suif library suif1	16,434	Y
	Suif library useful	18,022	Y
Front-end	Suif scc/snoot (parsing)	14,540	Y
	Suif porky (lowering)	16,679	Y
	Suif linking (linksuif)	2,352	Y
	inlining	789	Y
	loop unrolling	6,470	Y
	call-graph	172	
Total	Locally developed code	40,784	

Table 2.1: *C++ source lines of code in CASH v2.0. "Imported" code was developed elsewhere.*

- `gcc` does some extra work compared to CASH in this setup. It performs parsing (which in CASH is done by the Suif front-end)[11], instruction selection, register allocation and scheduling.

- We only display the programs for which CASH has compiled all functions. As described in Section 2.5.3, functions with variable arguments or functions calling `alloca` were excluded from the CASH compilation.

- We also compile some FORTRAN programs after translating them to C with `f2c`.

- CASH takes advantage of the call-graph to skip compilation of the unreachable functions.

Figure 2.6 shows that CASH is a relatively slow compiler. On average it is about 11 times slower than `gcc`, but it can be as much as 137 times slower! Looking at the breakdown of the compilation time, we notice that the three slowest programs (`168.wupwise,099.go` and `300.twolf`) spend a lot of time in Boolean minimization. In fact, the `168.wupwise` outlier spends almost all its Boolean minimization in processing a single moderately large Boolean expression with 15 inputs and 21 outputs. In general the Boolean minimizer Espresso, which we are using, (which has been designed for hardware logic minimization), has behaved very well by minimizing much larger formulas quickly.

The next two slowest programs are two different versions of `vortex`, `147` from SpecInt95 and `255` from SpecInt2K. Both these programs spend an inordinate amount of time in parsing the Suif intermediate file before even reaching the build phase (`129.compress` also spends more than half of its time in parsing). The abnormal parsing times are due to the inefficient data structures and procedures used by Suif for merging the symbol tables of several files.

[11] However, even parsing the Suif intermediate files is a very time-consuming operation, sometimes much slower than parsing the original text files. It is unlikely that parsing unjustly penalizes `gcc`.

Finally, the sixth worst program (`301.apsi`) spends one third of its time in pointer analysis.

If we discount these extreme examples, for all programs most of the time is spent in optimization. We have not explored trade-offs between compilation time and code quality.

Chapter 3

Pegasus: a Dataflow Internal Representation

3.1 Introduction

The original motivation for this research was the creation of a C compiler, CASH, which translates programs into hardware circuits. Pegasus is the internal representation of CASH. During the compiler development the intended target has guided many of the design decisions. We think that it is easier for the reader to follow the details of this representation if the explicit connection to actual hardware is kept in mind. Probably the most important feature of Pegasus from this point of view is its *completeness* — the representation is self-contained, enabling a complete synthesis of the circuits, without requiring further information. Internal representations with this property are called *executable* [PBJ$^+$91]. The naming suggests that one can write an interpreter for executing such a representation.

Three important computational paradigms are brought together in Pegasus: (1) executable intermediate representations; (2) dataflow machines; and (3) asynchronous hardware circuits. Indeed, the most natural way to implement a custom dataflow machine (i.e., one executing a fixed program) is by using asynchronous hardware. Both the dataflow machine operations and the asynchronous hardware computational units start computing when they receive their input operands and both use fine-grained synchronization between data producers and consumers to indicate the availability and consumption of data. Pegasus is original because it is used for compiling an imperative language, such as C, into dataflow machines. The overwhelming majority of the work in compilation for dataflow machines dealt with functional, single-assignment languages. On the other hand, asynchronous circuits are customarily described in some form of Communicating Sequential Processes (CSP). Both these classes of languages are very different in nature from C.

Certainly, the adequacy of dataflow intermediate forms for compiling imperative language has been noted before. In fact, many of the Pegasus features are inspired by Dependence Flow Graphs [JP93], which is a dataflow representation used to represent FORTRAN programs. The key technique allowing us to bridge the semantic gap between imperative languages and asynchronous dataflow is Static Single Assignment (SSA) [CFR$^+$91]. SSA is an intermediate representation used for compiling imperative programs in which each variable is assigned only once. As such it can be seen as a functional program [App98]. Indeed, Pegasus represents the scalar part of the computation of C programs as a functional, static-single assignment representation.

The **main contribution** of this work in the domain of compiler intermediate representations is the seamless extension of SSA to incorporate other modern features of compiler representations, such as a representation of memory dependences, predication, and (forward) speculation. While other intermediate program representations have each previously unified some of these aspects, we believe we are the first to bring all of them together in a coherent, semantically precise representation.

Pegasus contains a mixture of high-level and low-level primitives. The high-level subset is suitable as a framework for general compilation being more or less target-independent (i.e., it can even be used to compile for a regular processor). The vast majority of the optimization algorithms we present in this text operate on the high-level representation and therefore are applicable in traditional compilers as well. The low-level Pegasus features deal mostly with implementing procedure calls in hardware. Namely, the "code" to create, destroy and populate stack frames and the calling conventions are rather specific to application-specific hardware. Although, for completeness, we present in this chapter all these features together, the reader should remember that the high-level features of Pegasus constitute a very general program representation.

3.1.1 Summary of Pegasus Features

Pegasus integrates several important compiler technologies, drawing on their strengths:

Dependence representation: Pegasus is a form of program-dependence graph, summarizing data-flow, data dependence and control-flow information. Pegasus makes explicit *all* the dependences between operations (either scalar computations or memory operations).

Soundness: As any representation exposing parallelism, Pegasus is non-deterministic (i.e., it does not enforce a total execution order of the operations). However, it can be easily proved that Pegasus has the Church-Rosser property (i.e., confluence) — the final result of the computation is the same, for any legal execution order of the operations.

Precision: Pegasus' building blocks have a precise semantics, given in detail in Appendix B. The semantics is compositional. Moreover, a Pegasus representation is a complete executable form, which describes unambiguously the meaning of a program without the need for external annotations.[1]

Single-Assignment: Pegasus is a sparse intermediate representation, based on a (slightly modified) form of static-single assignment. As such, it shares with SSA the space and time asymptotic complexity, enabling fast and efficient compiler manipulations. Pegasus uses classical SSA to represent value flow within each hyperblock. The modification consists in the treatment of the inter-hyperblock flow. We place a merge operator (ϕ node) at each hyperblock entry point for each variable live at that point. While this modification theoretically has worse asymptotic properties, in practice it is completely acceptable.

Expressive: Pegasus enables very compact implementations for a large number of important program optimizations. In fact, most of the common scalar optimizations can be rephrased as simple term-rewriting rules operating on Pegasus graphs, implementing some of these optimizations on CFGs requires the computation of dataflow information.[2] This fact is important not because it saves the compiler the time to perform the dataflow analysis itself (for most of the optimizations only simple bit-vector, or other constant-depth lattice analyses are required). The importance is in saving the effort to *update* the dataflow information when program transformations are applied. Although research

[1] However note that Pegasus also includes symbol tables. Since symbol tables do not change through the compilation process, we will ignore them from now on.

[2] Unfortunately the term *dataflow* is overloaded; in this document it is used to denote both dataflow machines and dataflow analyses. We hope that the context is always clear enough to disambiguate its meaning.

has shown how to perform incremental dataflow updates, maintaining several pieces of dataflow information while performing complex program transformations is a daunting compiler-engineering task.

Verifiable: Due to the compositionality of the semantics, the correctness of many optimizations can be easily argued (especially the ones performing term-rewriting). If a program fragment is replaced by an equivalent one, the end-to-end program semantics does not change. An important piece of future work is to apply formal verification techniques to prove that Pegasus transformations preserve program semantics.

Compositionality also simplifies the presentation of the examples in this text. We can unambiguously isolate contiguous subgraphs of a larger code fragment independent of their environment, since the semantics of the subgraph is well-defined.

Predication and speculation: These are core constructs in Pegasus. The former is used for translating control-flow constructs into dataflow, and the latter for reducing the criticality of the control-dependences. Both these transformations are borrowed in their form used in the compilation for EPIC architectures. They are effective in boosting the exposed instruction-level parallelism. Both predication and speculation are employed in Pegasus at the hyperblock level. Speculation is therefore only forward speculation, which evaluates all outcomes of forward branches within a hyperblock.

The rest of this chapter is devoted to describing Pegasus in detail: its basic components, and the algorithm for constructing Pegasus from a control-flow graph. (An early version of this chapter was published as [BG02b], and abridged as [BG02a].) Some low-level features needed to translate Pegasus to a static dataflow machine are also presented. The next chapter of this document is devoted entirely to program transformations on Pegasus, customarily called program optimizations.

3.1.2 Related Work

Most of the work on compilation for dataflow machines has started with functional languages such as Id [AN90, SAJ96] or restricted subsets of array-based languages [Gao90]. Prior approaches for compiling C to dataflow machines considered only a subset of the language [VvdB90]. To our knowledge CASH is the first compiler to handle the full language. Pegasus is a form of dataflow intermediate language, an idea pioneered by Dennis [Den74]. The crux of Pegasus is properly handling memory dependences without excessively inhibiting parallelism. The core ideas on how to handle this by using tokens to express fine-grained location-based synchronization was introduced by Beck, et al. [BJP91]. However, Beck's paper only considers reducible control-flow graphs and suggests the use of code restructuring to compile arbitrary programs. By using hyperblocks we elegantly solve this difficulty without any code overhead.

Our representation builds on Static Single Assignment [CFR$^+$91] and extends it to cope with memory dependences and predication. The explicit representation of memory dependences between program operations has been suggested numerous times in the literature: for example, in Pingali's Dependence Flow Graph [PBJ$^+$91]; or Steensgaard's adaptation to Value-Dependence Graphs [Ste95]. As Steensgaard has observed, this type of representation is actually a generalization of SSA designed to handle value flow through memory. Other researchers have explored adapting SSA for handling memory dependences, e.g.,[CG93, Gal95, LH98, CLL$^+$96, CX02, LCK$^+$98, LC97]. But we think that none of the approaches is as simple as ours, as shown by the simple semantics of the intermediate language that we offer in Appendix B. The presentation of the semantics is inspired by the one used in [PBJ$^+$91] for Dependence Flow Graphs.

The Program Dependence Web [OBM90] attempts to unify the representation of various forms of dependences. The algorithms described in that paper are complex, rely on reducible control flow graphs, and it is not clear whether they have ever been implemented. Gated-SSA [TP95] relies on an algorithm for optimal insertion of the Pegasus "control-flow" operators in arbitrary irreducible flow-graphs, but does not deal with memory dependences.

The integration of predication and SSA has been also made in PSSA [CSC$^+$99, CSC$^+$00]. PSSA however does not use ϕ functions and therefore loses some of the appealing properties of SSA. The methodology of using the hyperblock as a basic unit and the algorithm used by CASH for computing block and path predicates are inspired from this work.

The suitability of SSA- and dependence-based representations for simple dataflow analysis was noticed in [JP93]. On the other hand, research on adapting dataflow analyses to predicated representations has produced some complex algorithms: [GJJS96, KCJ00]. We integrate these two approaches very cleanly without any overhead. In fact, our dataflow analyses are as simple as the ones in [JP93], despite operating on predicated code.

3.2 Pegasus Primitives

In this section we describe the basic building blocks used by the Pegasus program representation.

3.2.1 Basic Model

Pegasus is a directed multi-graph whose nodes are operations and whose edges indicate value flow. A node may fanout the data value it produces to multiple nodes. The fanout is represented in the figures as multiple graph edges, one for each destination. An edge runs between the producer of the value and the consumer of the value.

The data is produced by the source of an edge, transported by the edge and consumed by the destination. Once the data is consumed, it is no longer available. In general, nodes need all their inputs to compute a result, though there are some notable exceptions.

Each node performs a computation on its inputs to produce its outputs. We assume the existence of primitive nodes for all the major arithmetic and logic operations. We also have nodes which represent constant values and some special nodes for complex operations, such as memory access or procedure calls. *Argument* nodes denote procedure arguments.

All scalar values are sent from node to node on the edges. Pointer and array accesses are made using memory access nodes which perform updates/inspections of an imperative memory store. One can give several interpretations to a data edge and to a computation. The model we have targeted is *static dataflow*: one edge can hold at most one value at a time.[3]

We believe that keeping in mind the hardware target model of Pegasus can ease the understanding of the representation. One should therefore see each point-to-point edge as a bundle of three uni-directional wires as in Figure 3.1.

- A "data" wire, carrying the information from the producer to the consumer.

[3] Pegasus can be easily adapted for targeting other models, such as (1) FIFO dataflow, where each edge can hold an unbounded number of values and delivers them to the destination strictly in the order they have been inserted by the producer; or (2) tagged-token dataflow, in which each edge can hold multiple values, each tagged with an "iteration instance." Static dataflow is more appropriate for a direct hardware implementation.

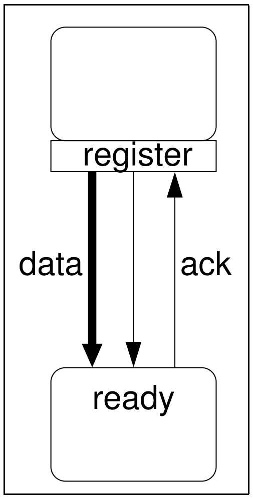


Figure 3.1: *Signaling protocol between data producers and consumers.*

- A "data ready" wire, also from producer to consumer, which indicates when the "data" wire can be safely read by the consumer.

- An "acknowledgment" wire, going from the consumer to the producer, indicating when the value has been read and therefore the "data" wire can be reused.

When there are multiple consumers, the bundle will contain one acknowledgment wire from each of them (see Figure 6.1). This signaling method is widely employed in the world of asynchronous circuits and is known as "bundled data protocol."

We will frequently use the term "wire" for such a bundle. Notice that data wires have a broadcast nature — they can have a fanout larger than one when the source value is used by several consumers. Physically we see this as a single bus, with one writer and many readers. In the figures a wire is represented as multiple graph edges, one for each consumer.

If the graph nodes should be thought of as small arithmetic units, corresponding to the program computations, the closest approximation for the program (scalar) variables in Pegasus are the wires themselves. Multiple wires may correspond to a single variable. In fact, there will be exactly one wire for each assignment. If the original program is in the classical SSA form, then each scalar SSA-renamed variable corresponds unambiguously to a single wire. However, program transformations can destroy this one-to-one correspondence.

The single assignment nature of Pegasus translates in the hardware world as the property that each data wire has a single writer. This is important for the synthesized circuits since it means that there is no need for arbitration for sending data, making data transfer a lightweight operation.

3.2.2 Data Types

Pegasus does not do any interesting type-related transformations and therefore types are passive objects during compilation, and will not be mentioned further. Each wire has an associated type, indicating the nature of the value transported. As a compilation invariant, all Pegasus representations are well-typed. All casts required for transforming the original program into a well-typed representation (according to the C standard conversion rules) are inserted by a preprocessing step, before the creation of the Pegasus representation begins. All Pegasus code transformations preserve types.

In the hardware view of the representation, the wire is a bus wide enough to represent all possible values of the data type. The BitValue program analysis algorithm [BSWG00] can be used to discover bits which do not carry useful information.

Pegasus programs manipulate the following types:

- All original scalar C types: `enum`, `signed` and `unsigned char`, `signed` and `unsigned short`, `signed` and `unsigned int`, `signed` and `unsigned long`, the gcc extension `signed` and `unsigned long long`, `float`, `double`, `long double`.
- All legal C pointer types.
- A Boolean type, requiring a single bit.
- A *token* type, used to serialize operations with side effects. Its usage is described in detail in Section 3.2.3.5.
- A *program counter (pc)* type. “pc” is actually a bad name choice for this type, whose values stand for the return address of functions (i.e., they represent the identities of the continuation nodes). Program counters occasionally need to be saved in memory to implement recursive procedures.
- A *current execution point* token. This value indicates which of the hyperblocks is currently executed. (An invariant is maintained that at any moment there is exactly one such value in the whole program.)
- Occasionally some of the wires can transport “wide” objects, such as structures, which in C can be passed by value and copied with a single statement. Currently we assume that these are implemented literally as very wide buses. But a practical hardware solution should involve some more complicated communication protocol.

In a traditional setting the “program counter” serves several purposes, which have been decoupled in Pegasus:

1. Its saved value at run-time is used to indicate return addresses. In Pegasus this purpose is taken by the “pc” type values.
2. It indicates the currently executing instruction. In Pegasus this is coarsely approximated by the “current execution point”, which indicates the currently active hyperblock.
3. It is used to give a “program order” for memory operations, which is used to limit the reordering of dependent instructions. This purpose in Pegasus is achieved by the tokens.

We use the following graphical convention in all the figures in this document:

- Solid lines represent regular program value flow.
- Dotted lines represent Boolean values, most often used to implement predicates.
- Dashed lines represent tokens or the current execution point.

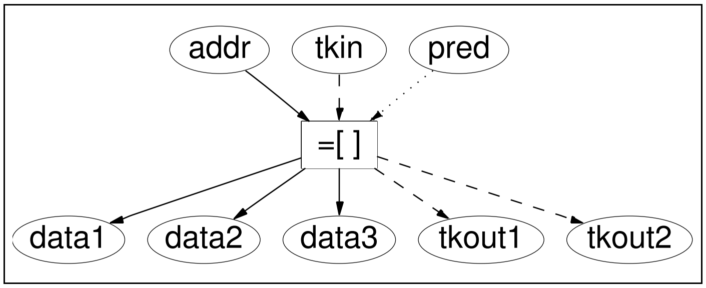


Figure 3.2: *A load operation with three inputs and two outputs. The data output (solid line) has a fanout of 3 and the token output (dashed line) has a fanout of 2. The three inputs are: data (solid line), predicate (dotted line), and token (dashed line).*

3.2.3 Operations

In this section we describe the complete assortment of computational primitives used in Pegasus to represent and manipulate programs. We resort here to an exhaustive description — the purpose of some of these operations may be cryptic until we show how it is actually used, later in this document.

Most operations have multiple inputs and some operations may produce more than one value. For example, a memory load operation produces not only the loaded data, but also a token indicating that memory access has completed. We will call the values produced by an operation its *outputs*. Do not confuse the number of outputs with the fanout of each output. An operation may produce two outputs (see one example in Figure 3.2), the first of which is consumed by three other operations, while the second of which is consumed by two other operations. We say that the fanout of output 0 is 3, while the fanout of output 1 is 2. Unfortunately, at the time of this writing, `dot` [GN99], the program which we use to draw most of these figures allows no control on the placement of the arrows. Therefore, in Figure 3.2 the two outputs with a total of five consumers are drawn as five different lines. This particular figure can be read unambiguously by knowing that for a load operation output 0 is data (solid line) and output 1 is token (dashed line), so each of the five lines can be uniquely attributed. In the `dot` drawings we use color to disambiguate inputs. Unfortunately colors cannot be reproduced clearly in this document and therefore the reader will have to cope with this ambiguity.

Usually the outputs are easy to tell apart. However, as Figure 3.3 illustrates, in general there will be no relationship between input order and the order the arrows are drawn in figures.

3.2.3.1 Arithmetic

We assume the existence of primitive operations for all essential arithmetic C operations, for all the legal type combinations. Table 3.1 gives their complete list.

3.2.3.2 Boolean

The complete list of operations producing Boolean results is given in Table 3.2.

3.2.3.3 Multiplexors

Like SSA, Pegasus has an explicit representation for indicating that different assignments of a variable are “merged”, i.e., they reach a same consumer. SSA uses for this purpose ϕ functions. In Pegasus however

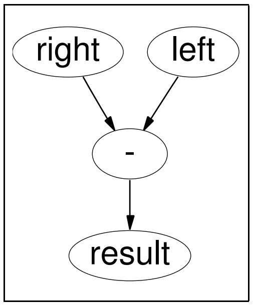


Figure 3.3: *A subtraction operation. Unfortunately the order of the arrows in the figure does not indicate which one is the left input and which is the right input.*

Name	**Symbol**	**Types**	**Observations**
Plus	$+$	arithmetic types	One input can be a pointer and other integral.
Minus	$-$	arithmetic types	One or both inputs can be pointers.
Negation	$-$	signed arithmetic types	
Noop		all types	Instantaneous propagation from in to out.
Register		all types	Delayed propagation from in to out.
Multiply	$*$	arithmetic types	
Multiply high	muluh	`signed`, `unsigned`	Upper 32 bits of a 32×32 bit product.
Division	/	all arithmetic types	Third input is predicate.
Modulo	%	integral	idem
Cast	(t)	all types (except structures)	
Equality	==	arithmetic types	Output is Boolean.
Inequality	!=	arithmetic types	idem
Less than	$<$	arithmetic types	idem
Less or equal	$<=$	arithmetic types	idem
Complement	˜	integral types	idem
Bitwise and	&	integral types	idem
Bitwise or	|	integral types	idem
Bitwise xor	ˆ	integral types	idem
Left shift	<<	integral types	idem
Right arithmetic shift	>>	signed integral types	idem
Right logical shift	>>	unsigned integral types	idem

Table 3.1: *All arithmetic operations used by Pegasus. The arithmetic types are:* `signed`, `unsigned`, `long long`, `unsigned long long`, `float`, `double`, `long double`.

Name	**Symbol**	**Observations**
Negation	!	
And	&&	Arbitrary fan-in allowed
Or	||	idem
Comparisons	see Table 3.1	Output is Boolean.

Table 3.2: *All operations producing Boolean values.*

there are two types of merge operators: mu (μ) which expect only one of the inputs to be present; and multiplexors, which expect *all* inputs to be present and dynamically select one of them. The latter are used

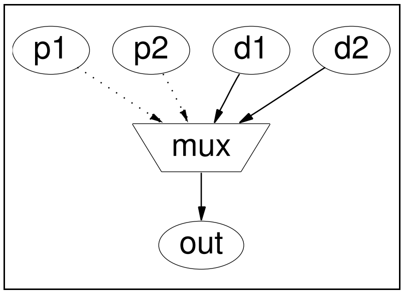


Figure 3.4: *Decoded multiplexors can have a variable number of inputs — each predicate input controls a data input. Unfortunately in the figures there is no easy way to tell which controls which.*

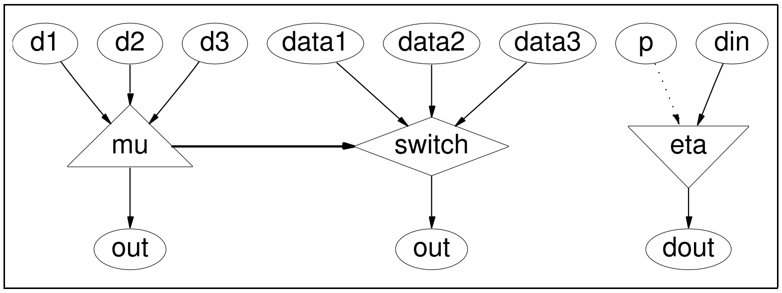


Figure 3.5: *Control-flow operations, from left to right: mu, switch, eta.*

to support speculation across all branches. We introduce the latter in this paragraph and the former in the next one.

Pegasus explicitly uses *decoded multiplexors* (they correspond to the γ operations in GSA, PDW [OBM90] and VDG). A decoded multiplexor selects one of n reaching definitions of a variable. It has $2n$ inputs: one data and one predicate input for each definition. When a predicate evaluates to "true", the multiplexor selects the corresponding input. The program transformations maintain the invariant that the predicates driving a multiplexor are always disjoint, making its behavior unambiguous. Note that it is possible for *all* predicate inputs of a mux to be simultaneously "false;" this happens when all input values are irrelevant for the result of the computation (i.e., they are generated on paths which are mis-speculated). The output value is then also irrelevant.

Although the decoded representation is somewhat redundant, its explicitness makes graph transformations during the optimization phase significantly easier to express. Unlike the SSA ϕ functions, the presence of the predicates explicitly encodes the path taken by each value to reach the merge point. Figure 3.4 shows a multiplexor representation.

3.2.3.4 Control-Flow: Eta, Mu and Switches

Pegasus makes use of three computational primitives borrowed from dataflow machines to steer data. They are used to transform control-flow into data-flow, as described in detail in Section 3.3. Figure 3.5 shows how they are represented.

Mu (μ)[4] operations together with multiplexors stand for all SSA ϕ functions. They must have n data inputs and one data output. At each moment at most one of the n data inputs should be available. That input

[4] We will occasionally use the term "merge" for "mu."

is copied to the output immediately. Mu operations are necessary to handle initialization of loop-carried variables but Pegasus uses them to model all inter-hyperblock transfers of data (see Section 3.3). Mu nodes can optionally have a second output, whose value is the *index* of the input that has last received data. We have depicted this second output with a bold line in Figure 3.5 — its only purpose is to drive "switch" nodes.

Switch nodes serve the same purpose as mu nodes, having n data inputs and one data output. However, they also have a *control input*, which indicates which of the data inputs is expected to fire next. They are similar to multiplexors, in the sense that the control input selects which of the data inputs to forward. Unlike multiplexors, they expect only one data input to be present (and not all of them). If a data input different from the one indicated by the control arrives, it is blocked (i.e., not acknowledged). Switch nodes are not used at all during program optimizations. They are used only by the back-end synthesizing a static dataflow machine from the program.

Eta (η) operations are also called gateways. The name "eta" was introduced in the Gated-Single Assignment form [OBM90]. They correspond to "T-gates" in the dataflow model. Each eta has two inputs: one for data and one for a predicate. There is only one output. The η is the only node which can consume inputs without producing outputs. It works in the following way: if the predicate is true, data is forwarded to the output; if the predicate is false, data is simply consumed (acknowledged) and nothing is forwarded.

When both the data and predicate inputs are compile-time constants[5], eta nodes have *three* inputs instead of two. The third input is the *current execution point*, as described in Section 3.2.2 (the *current execution point* can never be constant). The third input does not influence the computation at all — however its presence is required in order to generate the output. Without its presence, there would be no indication of *when* this value is needed.

3.2.3.5 Tokens

There is no clear notion of "program counter" for Pegasus programs. Execution happens immediately on availability of data, therefore being *dependence-driven*. However, not all program data dependences are explicit. Operations with side-effects can interfere with each other through memory. To ensure correctness, the compiler adds edges between those operations whose side-effects may not commute. Such edges do not carry data values. Instead, they carry an explicit synchronization *token*, an idea borrowed from Dependence Flow Graphs [BJP91]. Operations with side-effects wait for the presence of a token before executing. On completion, these instructions generate an output token to enable execution of their dependents. The token is really a data type with a single value, requiring 0 bits of information. In hardware it can be implemented with just the "data ready" signal. At run-time, once a memory operation has started execution and its execution order with respect to other side-effecting operations is *guaranteed*, the operation generates tokens for its dependents. The token can then be generated *before* memory has been updated. See Section 6.5.2 for more details on the run-time behavior of tokens.

The operations with memory side-effects (loads, store, calls, returns, stack frame manipulations, all described below) have a token input. When a side-effect operation depends on multiple other operations (e.g., a write operation following a series of reads), it must collect one token from each of them. For this purpose a *combine* operator is used. A combine has multiple token inputs and a single token output — the output is generated after it receives all its inputs. The combine was called "synch" in [BJP91]. Figure 3.6(a) shows a token combine operator, depicted by "V" — dashed lines indicate token flow and boxes indicate

[5] In this case the predicate must be constant "true" because etas controlled by constant "false" predicates are deleted by an optimization pass described in Figure 4.5.

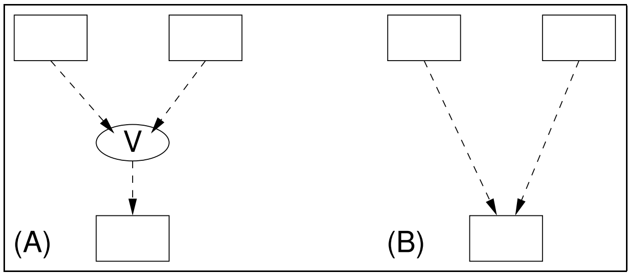


Figure 3.6: *(A) "Token combine" operations emit a token only on receipt of tokens on each input. (B) Reduced representation of the same circuit (used only to simplify figures) , omitting the token combine.*

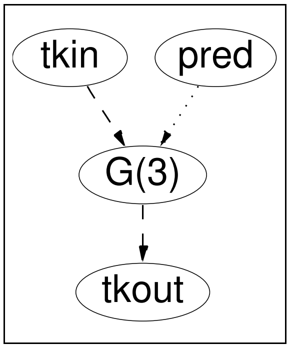


Figure 3.7: *A* token generator *G has a token and a predicate input and a token output. Its behavior is characterized by a constant parameter whose value is 3 in this figure.*

operations with side-effects. In some figures, to reduce clutter, we elide combines without introducing ambiguity, depicting Figure 3.6(a) as in Figure 3.6(b).

Token edges explicitly encode data flow through memory. In fact, the token network can be interpreted as an SSA form encoding of the memory values, where combine operations are similar to ϕ-functions [Ste95]. The tokens encode both true-, output- and anti-dependences, and they are "may" dependences.

Another operation manipulating tokens is the *token generator*, shown in Figure 3.7. A token generator has two inputs: a token and a predicate, and one token output. It also has a constant (compile-time) parameter n, the token reserve. n is a positive integer, modeling the *dependence distance* between two memory accesses (see Section 4.5.4). The token generator maintains a counter, initialized with n. On receipt of a "true" predicate, the generator decrements the counter, and if the counter is positive, it emits a token. On receipt of a token, it increments the counter. On receipt of a "false" predicate, the predicate input is not acknowledged until the token counter becomes n again.

3.2.3.6 Memory Access

Variables in the original program become wires in Pegasus. Such a transformation is applicable as long as all the updates to the variable are known. However, for updates made through pointers, this solution is no longer applicable.

Let us note that the compiler front-end lowers all memory accesses to pointer form. Accesses in arrays or structure fields are all transformed to pointer accesses. Therefore Pegasus deals with memory using only two operations: load and store.

Since by default all operations may be executed speculatively in Pegasus, operations with side-effects

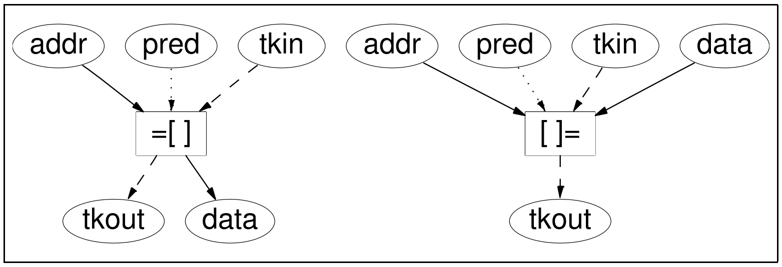


Figure 3.8: *Loads and stores. Besides address (and data for stores), they also require predicates and tokens as inputs. They both generate tokens at the output. The width of the data manipulated is indicated in some figures between the square braces.*

Operation	**Inputs**	**Outputs**
Call	procedure, n arguments, predicate, token, current	token
Continuation	token from call	returned value, token, current
Return	returned value, program counter, current, token	—
Argument	—	procedure argument

Table 3.3: *Pegasus operations for handling procedure calls.*

need to be controlled by a predicate indicating whether they are safe or not. When the predicate is dynamically "false", the operation is not executed. A load with a "false" predicate input may generate an arbitrary value. Note that at run-time the arrival of a "false" predicate does *not* enable the token release. A token at the input is required for this purpose as well because the emission of a token implies that all prior dependent operations have executed in the right order (see also Section 3.3.9.2).

Occasionally, to simplify some figures, we will explicitly indicate within a load or store operation the address accessed as a symbolic expression (see for example Figure 3.15).

3.2.3.7 Procedure Call and Return

Currently procedure invocation is a high-level primitive in Pegasus, with no low-level counterpart. However, stack management is explicit, as described below. For program optimizations this representation is sufficient. However, for mapping to real hardware a more detailed solution has to be crafted. The low-level solution is complicated by the need to support four arcane C features: (1) functions with a variable number of arguments; (2) calls through function pointers; (3) recursion; and (4) call-by-value for structures and unions.

Four operations are provided for handling procedures, shown in Table 3.3. The call is predicated, like all other operations with side-effects. The return lacks a predicate because of the way the compiler structures the code: as described in Section 3.3, each procedure has a unique epilogue, and therefore exactly one return instruction. "Control-flow" enters the epilogue only if the return is to be executed.

Figure 3.9 illustrates the call-return process. Besides an input which indicates the identity of the callee, the call node also receives a token and the *current execution point* token, both of which are sent to the callee. The call also sends the identity of a *call continuation* node, which will be the target of the return instruction.

Each procedure has a preamble containing *argument* nodes. When a call operation is executed, each of its parameter inputs is sent to a corresponding procedure input node. Besides the declared C procedure arguments, each procedure has three extra argument nodes: (1) the identity of the continuation node (i.e., the

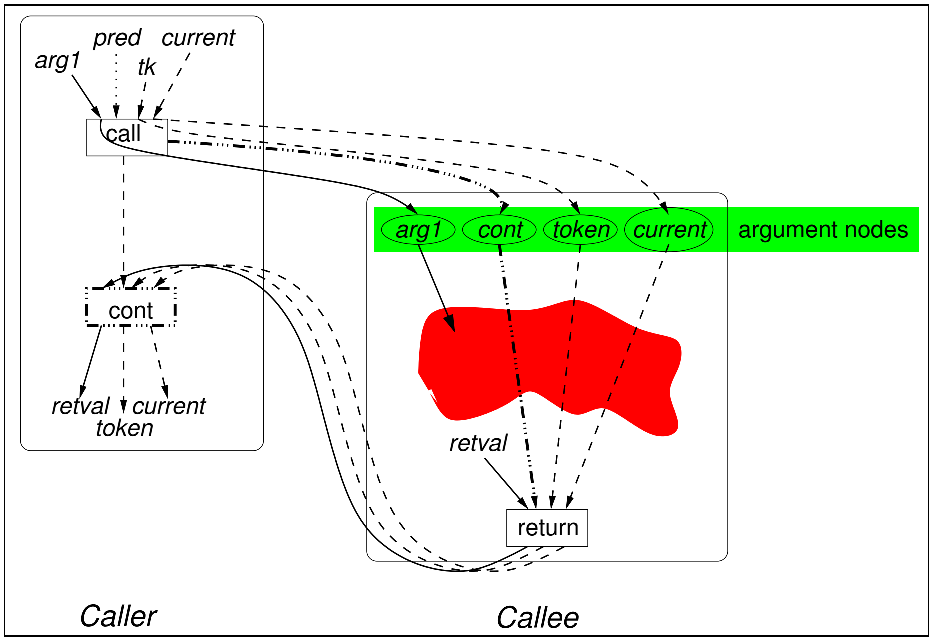


Figure 3.9: *Information exchange during the invocation of a procedure with only one argument* `arg1`*. Note that we have used a dashed-dotted line to indicate the fact that one of the arguments passed is the identity of the continuation node. This identity is used by the return instruction to route the returned value* `retval`*. This diagram shows the actual information flow at run-time and not the Pegasus representation. In particular, Pegasus does not represent any inter-procedural edges.*

program counter in the caller) where the return has to send the final result; (2) a token for serializing memory accesses; and (3) a *current execution point* input which effectively transfers control to the procedure.

The return instruction has four inputs, one being optional: the returned value (which is missing for procedures returning `void`). The other three arguments are a token, indicating completion of side-effects in the procedure, the *current execution point* and the identity of the continuation node. Finally, the continuation node receives from the return node the returned value, the token and *current execution point* and injects them back in the caller.

Due to separate compilation, the caller and callee may not be available simultaneously to the compiler. As a consequence, Pegasus does not include any edges spanning multiple procedures, i.e., the communication caller-callee is not explicitly represented. The representation contains however a “fake” token edge between the call and the continuation node, illustrating the fact that at run-time apparently execution is transfered directly from the call to the continuation node. However note that for recursive procedures the “call” node may be invoked multiple times before the “continuation” node gets control.

3.2.3.8 Constants

Nodes representing constants deserve special attention. There are three types of “constant” nodes in Pegasus, described in Table 3.4.

Constant nodes represent a compile-time constant. They are characterized by the constant value and its type (i.e., signed 0 is different from unsigned 0). To enable efficient optimizations, mostly to help common subexpression elimination, constants are kept in a hash table and each constant is represented only once in

Operation	Description
Constant	Compile-time constant, characterized by value and type.
Uninitialized	Describes an uninitialized value, characterized by type.
Hold	Loop-invariant value, inputs: predicate, value; output: value.

Table 3.4: *Nodes representing constant values.*

a hyperblock. However, to reduce clutter in figures, we draw a different constant node for each destination of its output wire.

We can differentiate two types of constants: compile-time constants and link-time constants. The latter include, for example, global symbol addresses which are not known until link-time. Note that not all symbol addresses are constant: symbols that are allocated on the stack have addresses which are not run-time constants.[6] As an optimization, CASH globalizes local objects whose address is taken from non-recursive functions, therefore transforming their addresses into constants.

Occasionally the user may not initialize some variables, or some compilation algorithms may need to introduce uninitialized values (for example, see our new register promotion algorithm in Section 4.6). For representing these in the dataflow framework, CASH makes use of a constant-like operation which is called *uninitialized*.

Constants are somewhat tricky to implement and formalize in a dataflow model, since, unlike the other nodes, they cannot just “fire” when their input is ready because they have no inputs. Their output is therefore permanently ready. Probably the best solution is to actually give each constant one input coming from the *current execution point*. In this manner, the constant will produce a result only when the current execution point indicates its value is needed. We have adopted a somewhat less elegant solution (but more efficient in terms of hardware) in the current implementation. We assume that in the synthesized circuit, constants are not independent nodes but are part of the destination. For example instead of a node “2” input to a node “+,” one would have a specialized node “+2.” Therefore constants are folded by the circuit synthesis back-end into each of their destinations. A node having all inputs constant should be constant-folded. The only exception is the eta, which should not be constant-folded; etas with both predicate and data inputs constant have a third input, the *current execution point*. Constants also cannot directly feed “mu” operations. Such constants must instead *always* feed an “eta” operation, which is the source of the mu.

Finally, in order to make more efficient the treatment of loop-carried values that never change in a loop, Pegasus uses the “hold” operations. Holds can be simply implemented in hardware as a register. However, the difference from a “register” operation is that on each iteration the hold operation needs to generate a new “data ready” signal, to enable the non-loop-invariant operations to consume a new value. The hold operation works in the following way: it loads and stores into a register the first value it sees on the input. Then, for each “true” predicate (indicating that the loop has not yet completed), it outputs a data-ready signal. When seeing a “false” predicate (indicating that the loop has completed), it resets the data in the register and waits for a new data item at the input.

3.2.3.9 Stack Frame Manipulation

The last set of operations we present, summarized in Table 3.5, deals with creation and destruction of stack frames for local per-procedure storage. In languages like C, stack frames have two purposes which are usually coupled: to hold caller/callee-saved registers, and to hold local variables which cannot be allocated

[6] Unfortunately Suif represents these as constants as well, making the translation from CFG somewhat more involved.

Operation	Inputs	Outputs	Observations
Create	current	address, token	Allocate a symbol; used internally.
Delete	address, token	token	Destroy a symbol; used internally.
Frame	token	frame address, token	Create a fresh stack frame.
Pop	token	token	Destroy last stack frame created.
Sp	token	frame address	Get current frame address (stack pointer).

Table 3.5: *Pegasus operations for manipulating the run-time stack.*

in registers (e.g., local variables whose address is taken). The CASH compiler completely separates these two usages of the stack. Since in Spatial Computation there are no shared registers between different procedures, only recursive procedures need to save registers. The discipline is caller-save only, as described in Section 3.3.11.

Stack-allocated locals are handled in two phases. First, *create* and *delete* operations are inserted to create and destroy each variable. This simplifies dataflow analyses for these variables, making the lifetime of each of them explicit. A *create* operation is parameterized with the size and type of the object to allocate. It receives as inputs the *current execution point* token and generates a new memory region, returning as outputs its base address and a token for serializing accesses to this region (it is known that the frame is disjoint from any other live memory region, so a new token is generated for serializing its manipulation). The *delete* operation is the dual: it receives the address and token and returns a token.

After optimizations for efficiency, all *create* operations of a procedure are lumped into a single *frame* operation and all *delete* operations are lumped into a single *pop*. These act similarly to create and delete. They are parameterized with the size of the frame to create/destroy. Therefore a single frame does the work of multiple create operations. The *sp* operation can be used to retrieve the base address of the last created stack frame.

* * *

This completes the tedious description of all Pegasus primitives. Most operations are straightforward. The use of the other operations will be hopefully made more clear in the next section, detailing an algorithm for creating Pegasus representations starting from traditional control-flow graphs.

3.3 Building Pegasus Representations

In this section we present an algorithm for constructing a Pegasus representation starting from a conventional control-flow graph (CFG). There are several reasons for presenting the Pegasus construction starting from a CFG rather than directly from source-level:

- Unlike source-level programs, CFG representations are “portable” across several languages.
- The semantics of a CFG is much simpler than that of a source language.
- As described in Chapter 2, our prototype compiler CASH uses Suif as a front-end for parsing and generating a CFG; the CFG is used as a basis for building Pegasus.

3.3.1 Overview

CASH performs almost exclusively intra-procedural optimizations. Therefore each procedure is parsed, optimized and generated independently. Pegasus contains no inter-procedural information. The only piece of inter-procedural information used by CASH is a conservative approximation of the program call-graph. The call-graph is used to identify and treat specially non-recursive procedures — in ASH recursion is rather heavyweight and therefore it is avoided if possible.

The construction of the Pegasus intermediate representation from a program is achieved in several stages: the program is parsed and translated to a CFG by the front-end. Next the CFG is decomposed into large fragments called hyperblocks, (Section 3.3.3), some of which will be the innermost loops. Each fragment is compiled into speculatively executed code, in order to extract the maximum ILP. For this purpose the control-flow inside each fragment is transformed into data-flow (Section 3.3.4) and branches are transformed into multiplexors (Section 3.3.5), resulting in an SSA representation. In order to maintain the correct program execution, predicated execution is used for the operations that cause side-effects or exceptions (Section 3.3.7). Operations with side-effects are serialized according to dependences using token edges (Section 3.3.9). The hyperblocks are linked together into a complete Pegasus graph by using special control-flow operators (Section 3.3.8). Structure is normalized by creating unique procedure prologue and epilogue sections (Section 3.3.10) and by handling the stack for implementing recursion (Section 3.3.11).

The various algorithms and data structures that we describe have the following asymptotic complexities:

- The complexity of the entire construction algorithm is given by the complexity of the component phases, the most complicated of which is a dataflow analysis used to determine points-to sets. Given the results of this analysis, constructing the intermediate representation (IR) takes linear time in the size of the resulting structure.

- The Pegasus graph is a sparse representation: its space complexity is similar to a classical SSA representation.

- Many of the optimization algorithms we present in Chapter 4 have linear time and space complexity in the size of the Pegasus representation. We discuss this aspect further when presenting the algorithms.

3.3.2 Input Language: CFGs

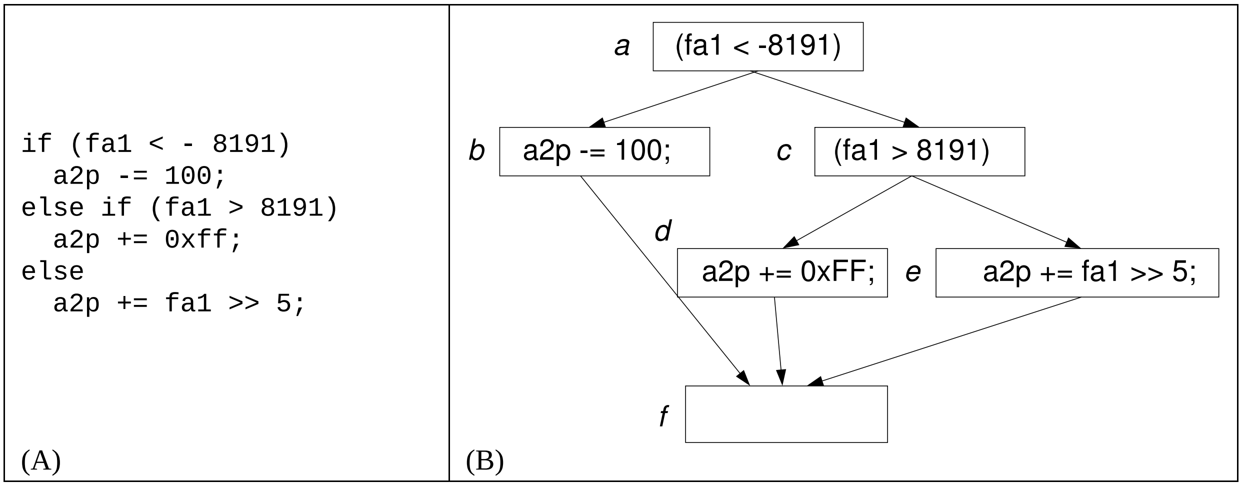
First let us briefly review the notion of control-flow graph. The input to the CASH compiler is a CFG, as represented by the Suif compiler. The atomic components of a CFG are three-operand instructions: each has two sources and one destination. The sources and destinations are either program variables or symbolic registers. Two special instructions, load and store, are used to denote memory accesses.

Instructions are grouped together into *basic blocks*, which are maximal contiguous program fragments which always get executed together. Each basic block starts with a label and ends with a branch instruction. In our illustrations we will represent CFGs as directed graphs, basic blocks are nodes, while branches are edges connecting the source block with the destination block. Figure 3.10 shows a source-level program fragment and its associated CFG.

We should note two important peculiarities in the way the CFG used to build Pegasus is structured:

1. Each `return` instruction is in a basic block on its own.

2. Potentially recursive procedure calls start a new basic block.



```
if (fa1 < - 8191)
  a2p -= 100;
else if (fa1 > 8191)
  a2p += 0xff;
else
  a2p += fa1 >> 5;
```

Figure 3.10: *(A) A code fragment from the* `g721` *Mediabench program; (B) Its control-flow graph. Each basic block (labeled from a to f) contains just one statement (except branches and labels).*

3.3.3 Decomposition in Hyperblocks

Building the Pegasus representation of a procedure starts by partitioning the CFG into a collection of *hyperblocks*. Hyperblocks were introduced in compilers that targeted predicated execution [MLC$^+$92]. A hyperblock is a contiguous portion of the CFG, having a single entry point and possibly multiple exits. The whole CFG in Figure 3.10 is a single hyperblock. In Pegasus a hyperblock is the unit of (speculative) execution: once execution enters a hyperblock, all component instructions are executed to completion. We will come back to this aspect in Section 3.3.7.

As pointed out in the literature about predicated execution, there are many choices for hyperblock boundary selection. At one extreme, one can make each basic block a separate hyperblock. In this case there will be no speculation. Howeverm CASH is at the other extreme, attempting heuristically to minimize the number of hyperblocks covering the CFG.

Hyperblock selection is made by a depth-first traversal. Hyperblock entry points are created with the following rules:

- at CFG back-edge destinations
- at `return` instructions
- at recursive calls
- at targets of edges exiting natural loops
- at blocks having incident edges from multiple different hyperblocks

In the process of hyperblock formation, CASH performs no code duplication or loop peeling and currently does not use any profiling information. Note that this algorithm does not need to do anything special for irreducible CFGs. Since each basic block is a hyperblock, we are guaranteed that a hyperblock cover of an arbitrary CFG always exists. This algorithm turns each reducible innermost loop body in the CFG into a hyperblock (if the loop contains no recursive calls). Outer loops always span multiple hyperblocks. Since

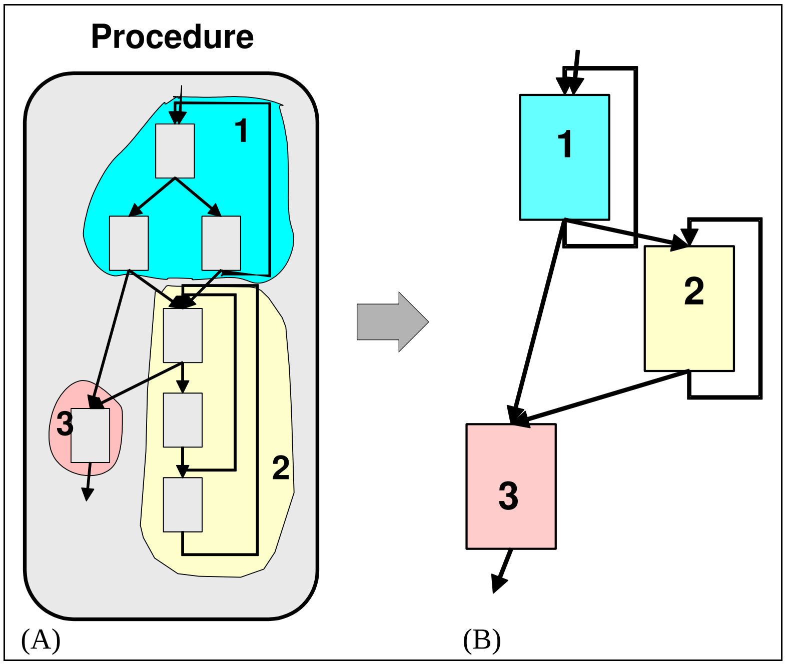


Figure 3.11: *(A) Sample CFG with (B) partition into hyperblocks. Hyperblock 2 contains multiple back-edges. Only cross-edges are shown in (B).*

most optimizations in CASH are at the hyperblock level, it is very desirable to have loops mapped to single hyperblocks. Although detecting outer loops using the hyperblocks structure is certainly feasible, it still requires some work. In contrast, hyperblocks containing innermost loop bodies are always specially tagged by the compiler and optimized specially. In hindsight we regret not having made outer loops first-class citizens.

After partitioning in hyperblocks we can classify each CFG edge as either *internal*, if its end-points are within the same hyperblock, or *cross-edge* when it connects two different hyperblocks. Technically, we will call all back-edges cross-edges, even though both their endpoints reside within the same hyperblock. The back-edges are the ones having as their target the hyperblock entry point. They are always loop-closing edges. A hyperblock can have not only multiple cross-edges, but also multiple back-edges as well. These can arise, for example, due to the use of C `continue` statements. Note that two different cross-edges may have the same destination. Figure 3.11 shows a sample non-trivial CFG and its cover with maximal hyperblocks according to the described algorithm.

Since they contain no loops, hyperblocks are directed acyclic graphs (DAGs). Then we can speak of the topological sort of a hyperblock with respect to the induced CFG.

3.3.4 Block and Edge Predicates

The next compilation step analyzes each hyperblock separately. It associates a *predicate* to each basic block and control-flow edge. These predicates are similar to the full-path predicates from PSSA [CSC$^+$00] and to the gating paths from GSA [OBM90, TP95]. Let us denote the set of CFG edges originating from a hyperblock by E (E comprises both internal edges as well as the cross-edges originating in the current hyperblock).

The predicate associated to basic block b, denoted by $p(b)$, is a run-time value which describes the condition under which b is on some execution path starting at the entry point of the hyperblock. In other words, b is (non-speculatively) executed when $p(b)$ evaluates to “true.” Similarly, the predicate $p(b, c)$ associated with a hyperblock edge (b, c) evaluates to “true” when the edge is on some non-speculative execution path starting at the entry point. Notice that these predicates are defined with respect to a hyperblock, and not with respect to the CFG.

The predicates are defined recursively, and can all be computed in $O(|E|)$ time in depth first order, as follows:

$p(\text{hyperblock entry}) = true$
 the hyperblock entry block is always executed (when the hyperblock itself is executed)
$p(x, y) = p(x) \wedge B(x, y)$
 edge (x, y) is executed if block x is executed and x branches to y
$p(y) = \vee_{(z,y)\in E}\, p(z, y)$
 block y is executed if some predecessor z branches to it

The predicates $B(x, y)$ correspond to the branch conditions in the code. Their computation is part of the computation of the basic block x. If x branches to y unconditionally, $B(x, y) = true$.

For the CFG in Figure 3.10, the predicates associated to the blocks and edges are:

$$\begin{array}{rcl}
p(a) & = & True \\
B(a,b) & = & (\texttt{fa1} < -8191) \\
p(a,b) & = & p(a) \wedge B(a,b) = B(a,b) \\
B(a,c) & = & \neg B(a,b) \\
p(a,c) & = & p(a) \wedge B(a,c) = B(a,c) \\
p(b) & = & p(a,b) \\
p(c) & = & p(a,c) \\
B(c,d) & = & (\texttt{fa1} > 8191) \\
p(c,d) & = & p(c) \wedge B(c,d) = B(a,c) \wedge B(c,d) \\
p(d) & = & p(c,d) \\
p(e) & = & p(c,e) \\
B(b,f) & = & B(d,f) = B(e,f) = True \\
p(b,f) & = & p(b) \wedge B(b,f) = p(b) \\
p(d,f) & = & p(d) \wedge B(d,f) = p(d) \\
p(e,f) & = & p(e) \wedge B(e,f) = p(e) \\
p(f) & = & p(b,f) \vee p(d,f) \vee p(e,f) = p(b) \vee (p(d) \vee p(e)) = p(b) \vee p(c) = True
\end{array}$$

Predicate expressions are simplified using algebraic manipulations and by exploiting the CFG structure. The latter simplifications are based on the fact (noted in [GJJS96]) that if basic blocks a and b are control-equivalent in the hyperblock-induced CFG, then $p(a) = p(b)$ (by definition, a is control-equivalent to b if a dominates b and b post-dominates a[7]). Notice that a and b can be control-equivalent in a hyperblock while not being control-equivalent in the procedure CFG. However, the reverse is always true. In Figure 3.10, blocks f and a are control equivalent and therefore have the same predicate. CASH carries a control-equivalence computation prior to computing predicates, and for control-equivalent nodes it always uses

[7] Recall that a basic block a dominates b if all paths from the start to b go through a. b post-dominates a if all paths from a to the exit of the procedure go through b.

the predicate of the earliest node in the topological sort because it will have the "smallest" formula, being dependent on fewer conditions.

The predicate computations are synthesized as *expressions explicitly represented in Pegasus*. In our figures edges carrying predicate values are denoted by dotted lines (see Figure 3.12).

3.3.5 Multiplexors

The next step in building the Pegasus graph is to convert each hyperblock into Static Single Assignment [CFR$^+$91] form, that is, to connect variable definitions to uses within hyperblocks. Instead of ϕ nodes, Pegasus uses multiplexors (see also Section 3.2.3.3). Unlike ϕ nodes, multiplexors explicitly represent the conditions under which each variable flows from a definition to a use, a method use in other intermediate representations such as GSA [OBM90], DFG [PBJ$^+$91], TGSA [Hav93], or VDG [WCES94].

Since hyperblocks are acyclic, the problem of mux placement is substantially simpler than the most general SSA case. The acyclic nature also makes the notion of controlling predicate for multiplexors well-defined. CASH uses a relatively simple implementation, which does not directly build a minimal SSA representation but adds multiplexors at all join points in the CFG. The insertion of muxes is preceded by a live variable computation. A mux will be inserted for variable x at a program point only if the variable is live there.

The mux insertion algorithm maintains for each basic block b and each live variable v the "last definition point" ***def***(b, v).

```
For each basic block $b$ in topological order
    For each variable $v$ live at the entry of $b$
        Create a mux $m(v)$
        def$(b, v) = m(v)$
        For each predecessor $b'$ of $b$
            Connect def$(b', v)$ to $m(v)$
            Set corresponding control predicate to $p(b', b)$
    For each instruction $i$ in $b$
        If $v$ is referred as a source in $i$, add edge def$(b, v)$ to $i$
        If $v$ is referred as a destination in $i$, def$(b, v) = i$
```

A subsequent multiplexor optimization phase (see Section 4.3 and Figure 4.4) removes the redundant multiplexors. In practice, the performance of our implementation is very good, especially since the complexity is bounded by the hyperblock size and not by the whole procedure CFG size. We have not found it necessary to implement the more efficient algorithm for multiplexor placement described in [TP95]. Figure 3.12 illustrates how edge predicates are used to drive the multiplexors for the example in Figure 3.10.

3.3.6 Predication

Traditional predication annotates each instruction in block b with the predicate $p(b)$. At run-time the instruction is executed if and only if its associated predicate evaluates to "true." In Pegasus, predicates become an additional input to each operation. The semantics of each operation is then slightly changed: if the predicate is "true", the instruction generates the expected output; if the predicate is "false", the instruction generates (immediately) an arbitrary value. In well-formed graphs the arbitrary value will be discarded by some multiplexor later in the code.

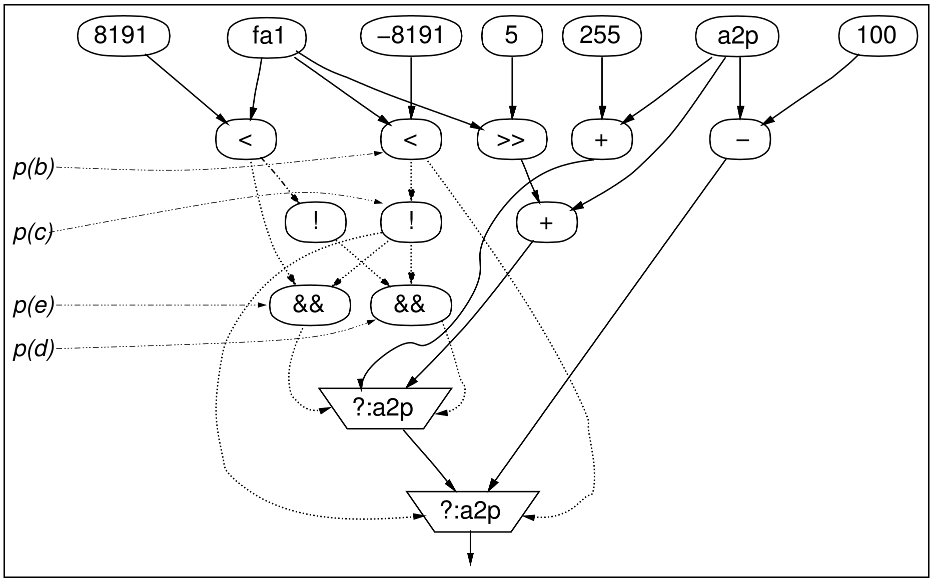


Figure 3.12: *Pegasus representation of the program fragment from Figure 3.10. The labels indicate to which basic block each predicates corresponds. The trapezoids are multiplexors. The dotted lines carry predicate values. This is the representation prior to multiplexor optimizations.*

3.3.7 Speculation

Speculative execution exposes ILP through the removal of control dependences [LW92] and reduces the control-height [SKA94] of the code. Pegasus aggressively exploits speculation for these purposes.

Predicate promotion, described in [MHM$^+$95], replaces the predicate guarding an instruction with a weaker one, causing the instruction to be executed even if the block including it is not. This is a form of control speculation, allowing the instruction to be executed before knowing the outcome of all controlling branches in the hyperblock (which have been folded into predicates).

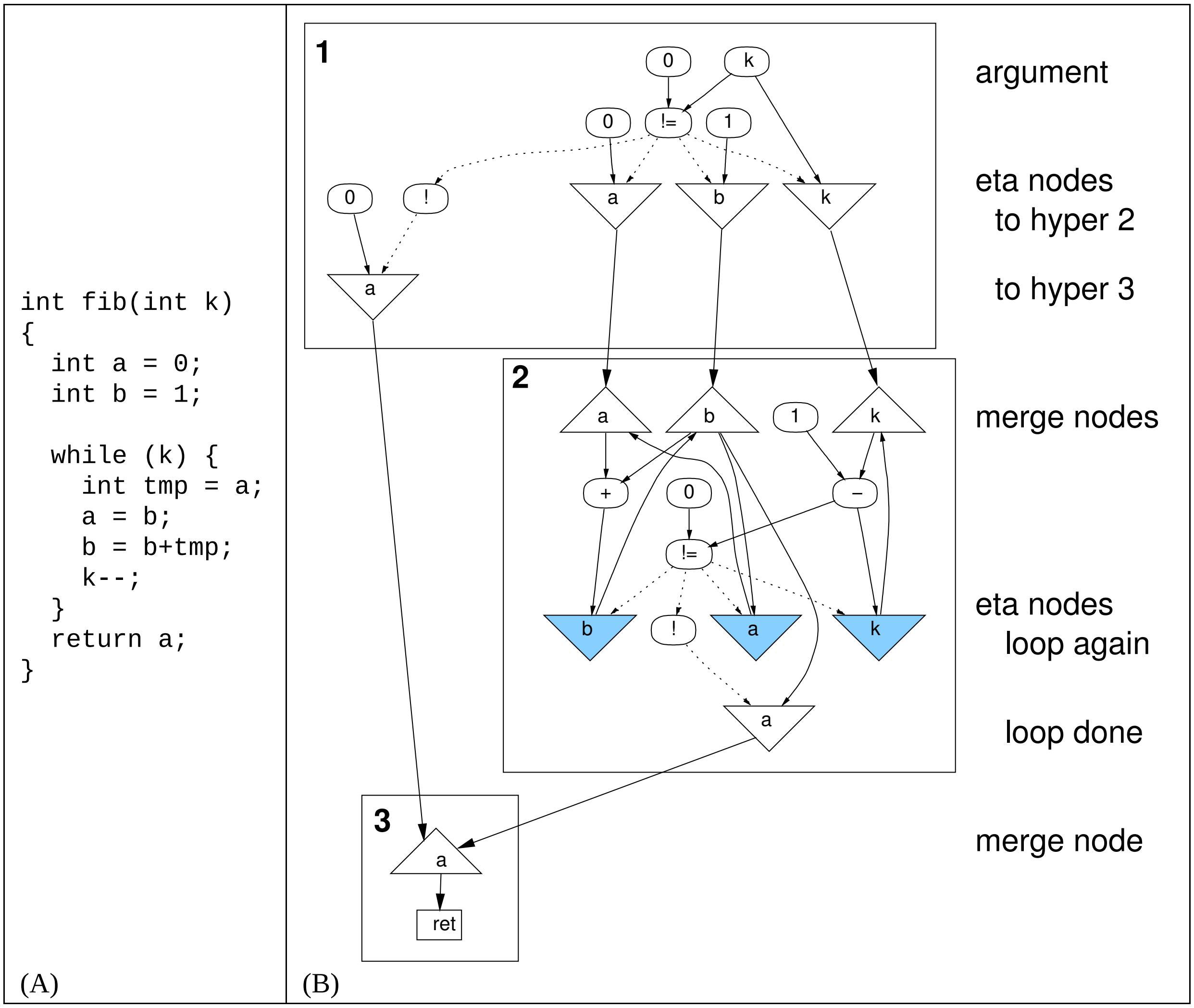
Pegasus promotes *all* instructions without side-effects by replacing their predicate with "true." This aggressive speculation enables many program optimizations because all the speculated instructions behave as if part of a large "straight-line" code region.

Some instructions cannot be speculatively executed. These are division (which can generate division-by-zero exceptions), control-flow transfers (e.g., inter-hyperblock branches, procedure calls) and memory operations (including also reads, since reading an illegal address can cause an exception). For these instructions we keep the controlling predicate unchanged. Since we use speculation, most instructions do not require a predicate input.

3.3.8 Control-Flow

The compilation phases described so far deal with hyperblocks in isolation. The next phase stitches the hyperblocks together into a graph representing the whole procedure by creating dataflow edges connecting each hyperblock to its successors. It therefore synthesizes objects corresponding to the CFG cross-edges. A special instance of cross-edges are loop back-edges, which are also synthesized in this step.

A cross-edges is synthesized for each live variable along that edge — each gives rise to an *eta*-node. Etas are depicted by triangles pointing down (see Figure 3.13 and Figure 3.5). Eta-nodes have two inputs—a



```
int fib(int k)
{
  int a = 0;
  int b = 1;

  while (k) {
    int tmp = a;
    a = b;
    b = b+tmp;
    k--;
  }
  return a;
}
```

Figure 3.13: *(A) Iterative C program computing the* k*-th Fibonacci number and (B) its (slightly simplified) Pegasus representation comprising three hyperblocks. This figure does not include the program counter, the token edges, or the* current execution point.

value and a predicate—and one output. When the predicate evaluates to “true”, the input value is moved to the output. When the predicate evaluates to “false”, the input value and the predicate are simply consumed, generating no output. Each eta is controlled by the edge predicate.

Hyperblocks with multiple predecessors may receive control at run-time from one of several different points. Such join points are indicated by *merge* nodes, shown as triangles pointing up.

Putting together all the transformations above, Figure 3.13 illustrates the representation of a program comprising three hyperblocks, including a loop. The eta nodes in hyperblock 1 will steer data to either hyperblock 2 or 3, depending on the test `k != 0`. Note that the etas going to hyperblock 2 are controlled by this predicate, while the eta going to hyperblock 3 is controlled by the complement. There are merge nodes in hyperblocks 2 and 3. The ones in hyperblock 2 accept data either from hyperblock 1 or from the back-edges in hyperblock 2 itself. The back-edges denote the flow of data along the “while” loop. The

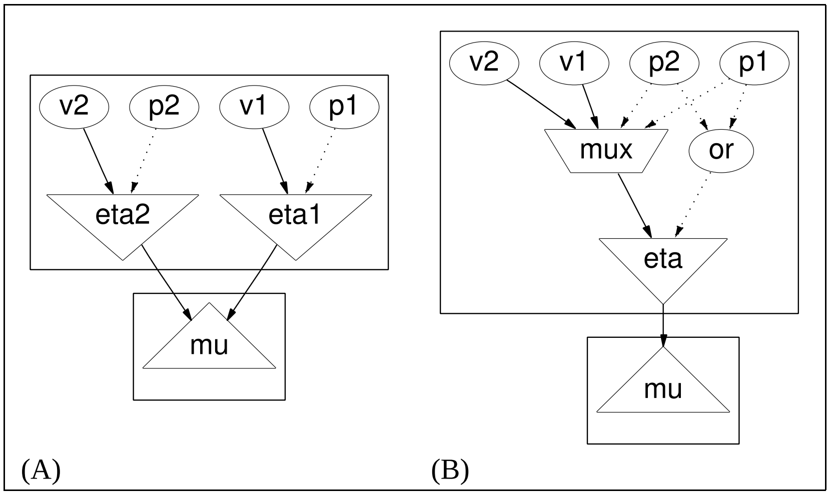


Figure 3.14: *Two possible implementations of cross-edges having the same source and destination hyperblocks.*

merge node in hyperblock 3 can accept control either from hyperblock 1 or from hyperblock 2.

The merge and eta nodes essentially transform control-flow into data-flow. The Pegasus IR tends to be more verbose than the classical CFG-based representations because the control-flow for each variable is represented separately. Notice for example how in hyperblock 2 of Figure 3.13 there are eta and merge nodes for all three loop induction variables, a, b and k. There are many benefits in this explicit representation, which naturally exposes parallelism.

3.3.8.1 Handling Back-Edges

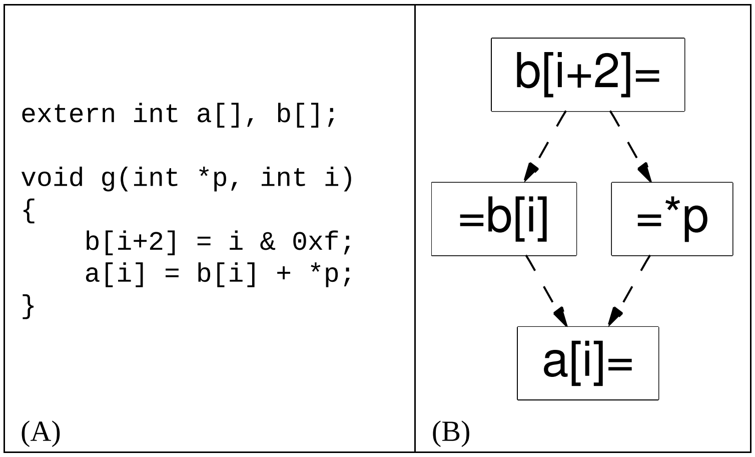
If multiple cross-edges have the same destination they can be handled in two ways: either use a separate eta for each of them, or build a multiplexor for all instances of a variable and use a single eta. Figure 3.14 illustrates the two alternatives. Functionally these two variants are equivalent, but the multiplexor may lengthen the critical path.

CASH uses both these representations: the multiplexor edges are used for implementing all back-edges, while the simple eta solution is used in all other cases.[8] The result is that for all hyperblocks there is exactly one back-edge for each live variable; in other words, for each mu/switch node there is exactly one back-edge eta. This invariant simplifies dramatically the implementation of some algorithms, such as induction variable detection.

3.3.9 Token Edges

As described in Section 3.2.3.5, interfering operations with side effects have to execute in program order. These operations are: loads, store, procedure calls, returns, operations for stack manipulation. The compiler therefore has to respect *dependences* between these. Dependences are first-class objects in Pegasus, being explicit inputs and outputs of operations.

[8] However, a compile-time option can be used to force the simple eta implementation for all cases.



```
extern int a[], b[];

void g(int *p, int i)
{
    b[i+2] = i & 0xf;
    a[i] = b[i] + *p;
}
```

Figure 3.15: *A program and the implementation of its memory synchronization, assuming all memory instructions can access any memory word.*

3.3.9.1 Token Edge Insertion

During the construction of Pegasus, token edges are inserted using the following algorithm:

- A flow-insensitive analysis determines local scalar values whose address is never taken. These are assigned to symbolic registers and become wires. In Figure 3.13 values `a`, `b` and `k` are scalars. All other data values (non-scalars, globals or scalars whose address is taken) are manipulated by load and store operations through pointers. This analysis and transformation is performed by the Suif front-end.[9]

- Each memory access operation has an associated read/write set [GH98] (also called "tags" in [LC97], M-lists in [CX02] or effects in [GJLS87]): the set contains the memory locations that the operation *may* access. This set can be computed through pointer analysis, or, if no pointer analysis is performed, it is approximated to contain the entire memory.

- Memory instructions are created by traversing the control-flow graph in program order.

- Instruction i receives tokens from all instructions j such that there is a control-flow path from j to i, and i does not commute with j (i.e., i and j are not both memory reads) and the read/write sets of i and j overlap. If an instruction receives tokens from multiple predecessors, a *token combine* operation is inserted.

Figure 3.15 shows a sample program and the subgraph induced by the token edges. Notice that the two memory reads (of `b[i]` and `*p`) have not been sequentialized, since read operations always commute.

The algorithm to insert token edges is somewhat similar to the one for multiplexor insertion. It is somewhat complicated by some special properties of pointer operations:

1. To synthesize the inter-hyperblock tokens, a scan is made of the whole hyperblock and all the objects modified through pointers are noted. Then a separate "mu" node is created for carrying a token for each of them.

[9] Using the command `porky -only-simple-var-ops`.

2. A pointer may access many values; therefore updates through pointers are in general non-destructive.

3. Since, in the absence of whole-program analysis, potentially each memory object is live at each program point,[10] each hyperblock has to carry tokens "covering" together the whole memory. For representing all "anonymous" objects (i.e., objects which are not accessed by name in the current hyperblock, such as heap objects or objects accessed through pointers passed as arguments), a special object called `nonlocals` is used.

4. The lack of inter-procedural analysis makes the treatment of procedures calls quite conservative.[11] Procedures are assumed to read and write all symbols whose address is accessible from the procedure arguments or globals.

5. The return instruction stands for a control-transfer to an unknown procedure, therefore it collects all the tokens before executing.

Figure 3.16 shows the token network created by application of these rules for a small program.

3.3.9.2 Transitive Reduction of the Token Graph

The graph formed by token edges is maintained in a transitively reduced form by the compiler throughout the optimization phases. This is required for the correctness of some of the optimizations described in Section 4.4. Since some graph transformations may break this property, the graph is repeatedly transitively reduced. The reduction is accomplished by scanning the subgraph induced by the token edges and removing the token edges whose destination is already reachable from the source. This operation is potentially very costly ($O(n^3)$), but the sparsity of the token graph makes in practice this canonicalization quite acceptable.[12]

The presence of a token edge between two memory operations in a transitively reduced graph implies the following properties:

- the two memory operations may access the same memory location, *and*

- there is no intervening memory operation affecting that location.

These properties are heavily used by our memory operation redundancy elimination (Section 4.4) and register promotion algorithms (Section 4.6).

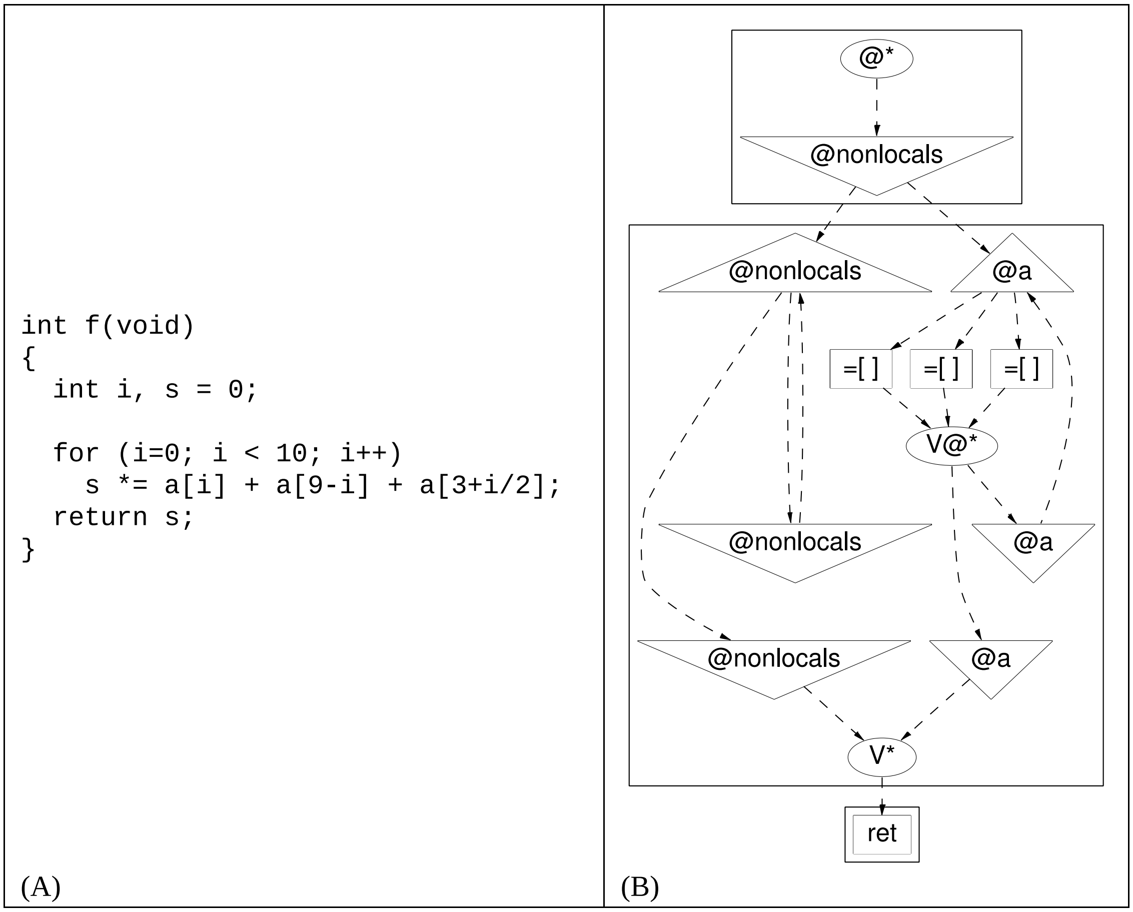
3.3.10 Prologue and Epilogue Creation

The unique "entry" basic block of a procedure always belongs to a loop-less hyperblock, used as the function prologue. This hyperblock contains the *argument* nodes, which stand for the procedure arguments. Code is synthesized in this hyperblock to create all stack-allocated local variables (these are all locals that may be accessed through pointers). For each such variable the compiler instantiates a *create* operation. This operation returns both the address of the variable, used for subsequent pointer manipulations, and a token mediating access to this variable.

[10] The only known information about lifetime available at this point is that local objects are dead at the `return` instruction.

[11] The lack of information on procedure calls is not an intrinsic feature of Pegasus, but of our current implementation. More refined information could be easily accommodated.

[12] One can construct examples which require dense (non-sparse) token-edge subgraphs to be accurately represented. In practice, however, the token edge subgraphs tend to be quite sparse. See Section 3.4 for an experimental quantification of this fact.



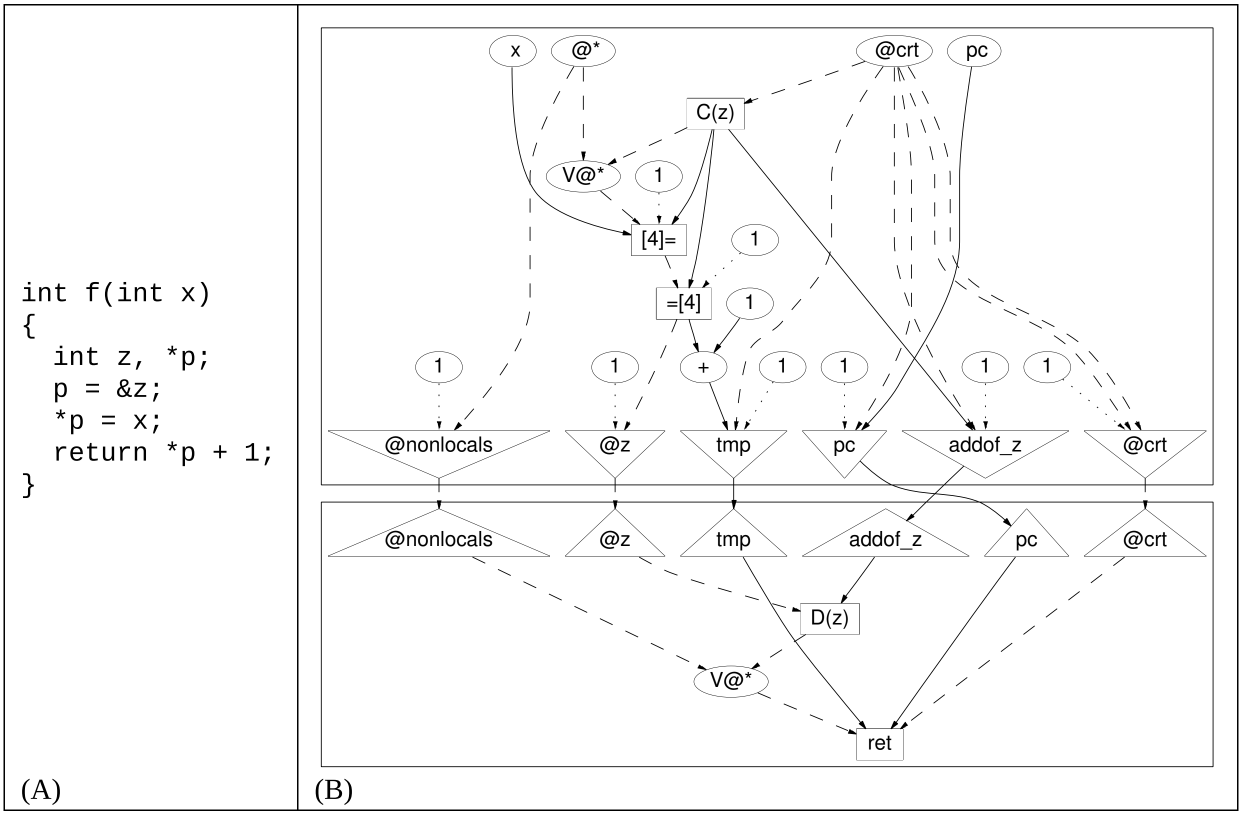
```
int f(void)
{
  int i, s = 0;

  for (i=0; i < 10; i++)
    s *= a[i] + a[9-i] + a[3+i/2];
  return s;
}
```

Figure 3.16: *(A) program manipulating an array and (B) the representation of its token network. The top node labeled "@*" is the procedure token argument. The nodes labeled with "@a" transport tokens mediating access to the array* `a[]`*, while the nodes labeled as "@nonlocals" transport tokens for all other memory locations.*

After the translation from CFG to Pegasus, a function containing many `return` statements contains one hyperblock for each of them. All these hyperblocks are identical. The epilog creation normalizes the graph by keeping only one of these and diverting all control-flow towards returns to go to the canonical return. Code is next inserted in the newly created epilog to handle the destruction of stack-allocated local variables: for each such variable a *delete* operation is created. The delete receives a token and the address, and destroys the local variable.

Optimizations may deem some stack-allocated locals useless (for example, after register promotion), and then the code manipulating them is simply deleted. Representing creation and destruction for each variable separately enables simpler dataflow analyses and optimizations. The compiler back-end efficiently synthesizes *create* and *delete* operations by bundling operations for all variables together in a single stack frame, whose allocation (using the *frame* operation) and destruction (using the *pop* operation) is more efficient.

Figure 3.17 shows a simple procedure and the synthesized representation. The resulting representation has exactly two hyperblocks: the prolog and the epilog. The prolog contains the inputs and the bulk of the



```
int f(int x)
{
  int z, *p;
  p = &z;
  *p = x;
  return *p + 1;
}
```

Figure 3.17: *Representation of a procedure containing a local variable whose address is taken. C(*z*) indicates a* create *operation which allocates variable* z*, and D(*z*) a* delete.

computation. Note the *create* operation, denoted by a C, which generates a new local (z). The outputs of the "create" are the address of z, denoted by addof_z, and the token mediating access to z, denoted by @z. The epilog contains the *delete* operation D followed by a return.

A discussion of the other nodes in Figure 3.17 is given below in Section 3.3.12. Figure 3.18 shows the final result of optimizing the example in Figure 3.17. The load from z is removed by register promotion (Section 4.4) and the *store* followed by the *delete* is deemed useless (Section 4.7.3). Next, since there is no access left to z (shown by the fact that the *delete* immediately follows the *create*), the local variable z is completely removed and this function requires no stack frame at all. While this particular example is very simple, such optimization occurs in real programs using local temporary arrays.

3.3.11 Handling Recursive Calls

The handling of recursive procedures is a low-level issue in Pegasus. ASH circuits have no centralized register file. All scalar data values are transported by "wires" directly between producers and consumers. If no procedure in a program is recursive, this model can be simply extended without the need of saving and restoring registers at procedure boundaries because each procedure has its own register set. The story is different when a procedure is recursive because a local variable may have simultaneously several live instances.

The policy in ASH is to leave the whole burden of saving "registers" on the caller. In order to simplify

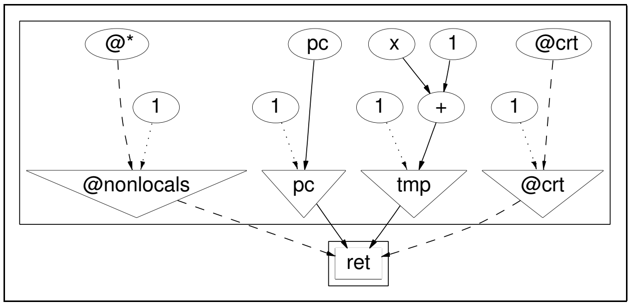


Figure 3.18: *The program in Figure 3.17 after all optimizations are applied.*

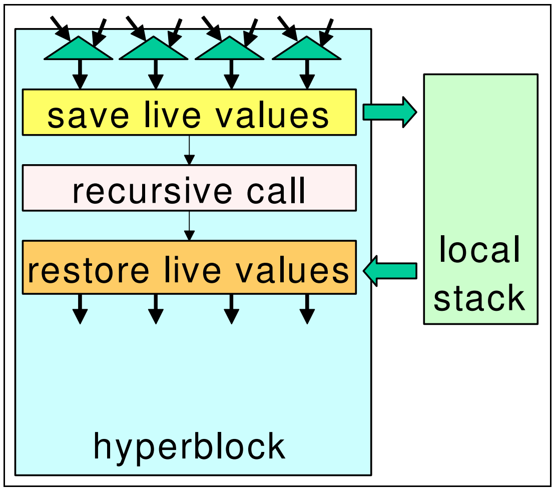


Figure 3.19: *Schematic implementation of recursion. The stack frame may be local or not. Currently the stack is co-located within the global memory space: this automatically handles stack overflow handling and passing the address of locals to other procedures.*

the task of deciding which values are live at the point of a recursive call, each such call must be the first instruction in a hyperblock. In C programs in general it is difficult to ascertain which procedures are recursive. Since the conservative solution of always saving all live values would entail substantial overhead, and worse, would break hyperblocks needlessly in too many pieces, CASH uses a conservative approximation of the call-graph to decide which calls may be recursive.[13] (The call-graph is also used to avoid compiling unreachable functions.)

Once a call is the first instruction in a hyperblock, the live values that need to be saved are all the hyperblock inputs, except the ones which are only passed as arguments to the call. Figure 3.19 shows this process schematically. All tokens are collected using a “combine” and passed as a single token input to the recursive call. Prior to making the call a stack frame is created (using the “frame” operation) for all live locals. The locals are stored in the frame prior to the call and restored on return. Figure 3.21 shows how the recursive procedure from Figure 3.20 is represented in Pegasus. The variables that need to be saved are `n` and `pc`. (In this example the `pc` is encoded in 12 bytes and `n` on 4, therefore the `frame` instruction allocates 16 bytes. `n` is saved at offset 0, while `pc` at offset 4.) Note that `tmp2`, whose value is `n-1`, is not live past the recursive call, and is not saved. Also, notice how the tokens for nonlocals and for the newly

[13] The call-graph is exact in the absence of calls through function pointers.

```
int rec(int n)
{
    if (!n)
        return 0;
    return
        n + rec(n-1);
}
```

Figure 3.20: *Sample recursive procedure.*

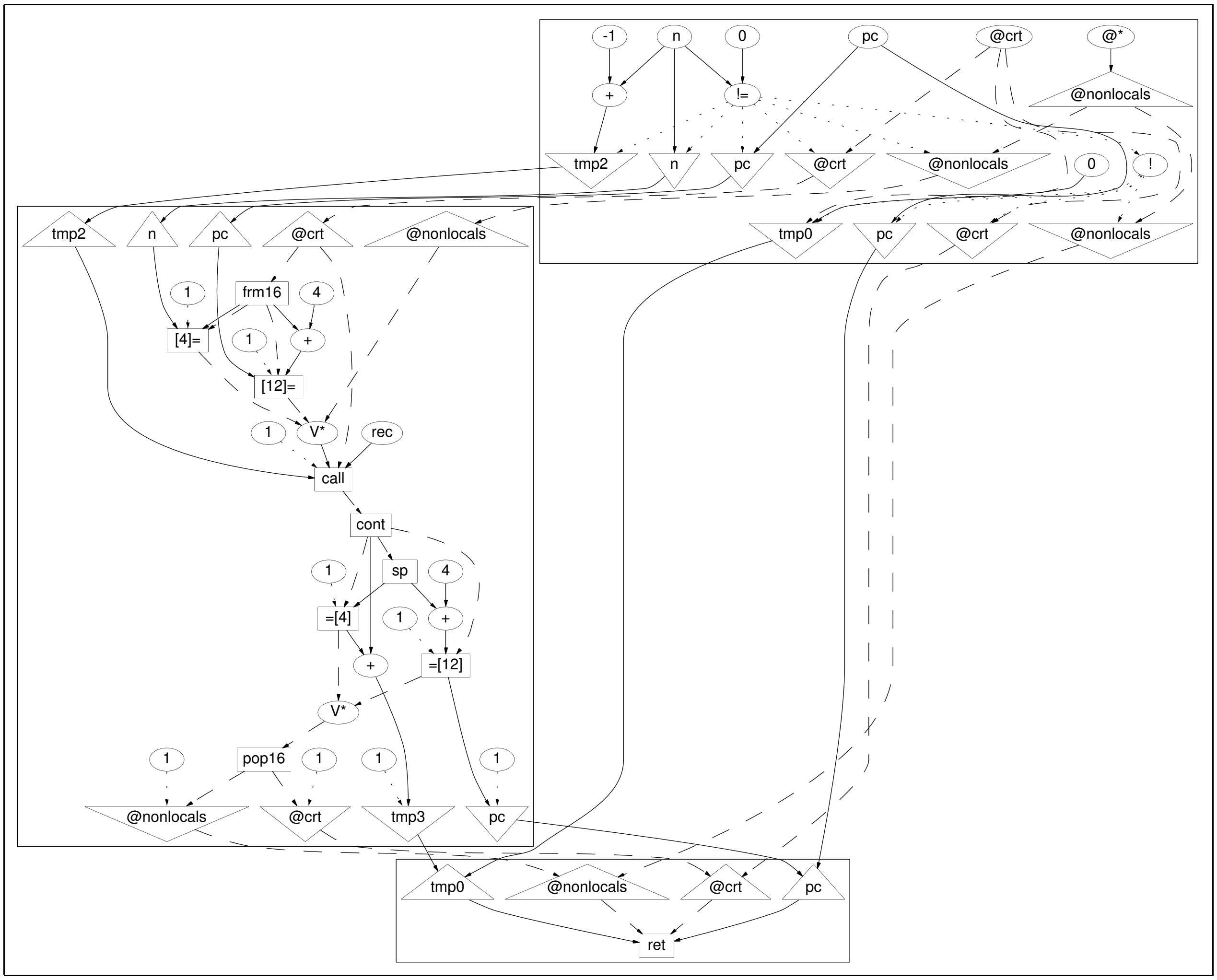


Figure 3.21: *Pegasus representation of the recursive procedure in Figure 3.21.*

created stack frame are combined before making the recursive call.

3.3.12 Program Counter, Current Execution Point

Figure 3.17 and Figure 3.21 show complete Pegasus representations. Most of the other figures in this document are simplified to highlight only the important features. In particular, in these two examples we show the flow of the values *program counter* (labeled `pc`) and *current execution point* (labeled `@crt`). These values are invariant in a procedure (can never change).

The `pc` holds the identity of the continuation node where the current procedure returns control on completion. How node names are encoded is not specified by Pegasus. A new `pc` is created by a `call` operation and is used by a `return`. Recursive procedures need to save the `pc` on the stack.

The `@crt` nodes and edges are used to propagate the so-called *current execution point* token. The invariant maintained at run-time is that there is exactly one such value “set” in the whole program. The hyperblock containing this value is sometimes called the “current execution point.” However, note that this notion is more than vague since computations can be carried simultaneously in multiple hyperblocks, or even procedures, at once. The `@crt` value has several important uses at run-time:

- It is used to drive eta nodes with all inputs constant.
- It is used to pass control from caller to callee and vice-versa.
- The *mu* nodes for `@crt` are used to drive the *switch* nodes (see Section 6.3.1).

3.3.13 Switch Nodes

In Chapter 6 we discuss some invariants maintained by the proper execution of ASH circuits. In Section 6.3.1 we show a scenario in which some of these invariants may be violated unless some extra care is taken. The fix for these problems is described in the same section and involves transforming some “mu” nodes into “switch” nodes. Currently this transformation is performed late in the compilation process. This is of some concern, since it implies that the representation up to that point does not accurately reflect the program semantics. Ideally “switch” nodes should be created right from the start. However, the compilation process is correct because all the optimizations we have implemented operate in the same way on “mu” or “switch” nodes. The reason is the fact that there is only a very limited set of code motion transformations applied (mostly within the same hyperblock), and the “switch” nodes only affect the flow of data between different hyperblocks.

3.4 Measurements

In this section we present static measurements on the program representation. While we have observed empirically that the trends presented in this section are the same for all programs we have compiled, we quantify them precisely here only for nine randomly programs from three of our benchmark suites, displayed in Table 3.6. The program size is just the raw number of source lines, counted with `wc`.

A first concern is the “verbosity” of Pegasus when compared to a traditional CFG. Figure 3.22 shows that for the selected programs the representation is indeed larger, but never more than two times as large, even if aggressive loop unrolling and procedure inlining are used in both cases. This overhead is quite acceptable. We are measuring the size of the Pegasus representation after all optimizations have been applied. Optimization can both increase the number of nodes (for example, expanding constant multipliers) and reduce it (for example, CSE or dead-code).

Benchmark suite	Program	Lines
Mediabench	adpcm_e	319
	epic_d	2,482
	mpeg2_d	9,844
SpecInt95	099.go	29,275
	124.m88ksim	19,168
	147.vortex	67,220
SpecInt2K	175.vpr	17,744
	197.parser	11,402
	256.bzip2	4,661

Table 3.6: *Selected programs for static measurements*

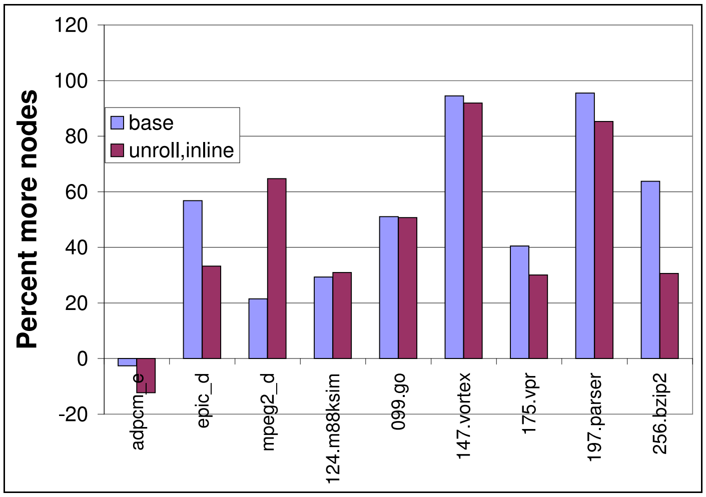


Figure 3.22: *How much larger a Pegasus representation is compared to Suif, with and without unrolling and inlining. The size is given in "nodes" for Pegasus and operations of the CFG for Suif.*

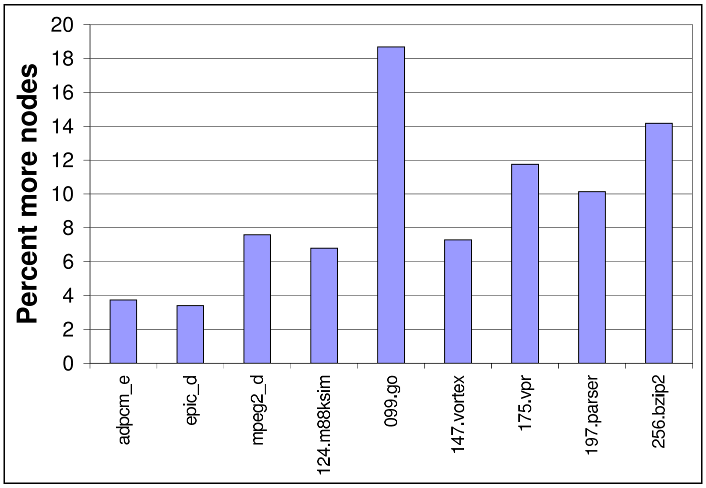


Figure 3.23: *Increase in representation size due to fine-grained memory dependence representation.*

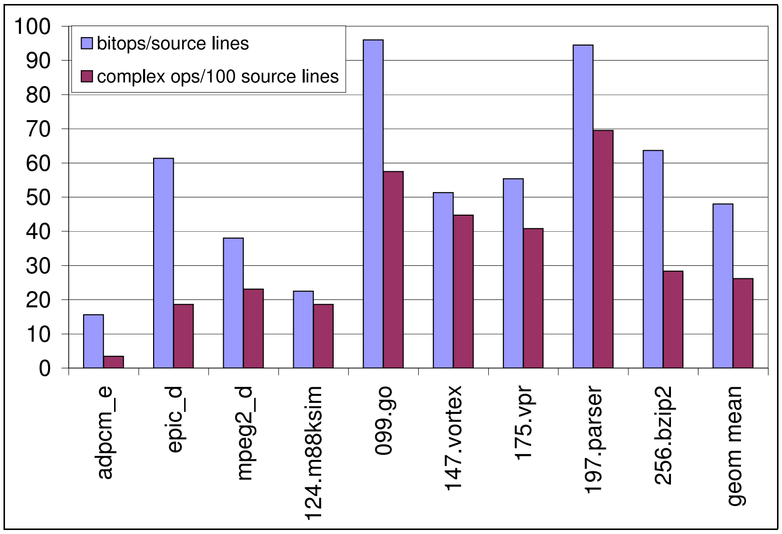


Figure 3.24: *Estimated hardware resources for implementing each the programs completely in hardware. The values are given in bit-operations per source line of code and complex operations per 100 lines of code. The data is obtained without unrolling or inlining.*

The space complexity of the scalar network of Pegasus (i.e., ignoring the memory tokens) is asymptotically the same as a classical SSA representation. There was some concern that explicitly representing may-dependences would blow up quadratically. However, independent of which memory optimizations were turned on or off, the size of the IR never varied by more than 20%, as shown in Figure 3.23.

We have also attempted a back-of-the-envelope calculation of the required hardware resources for implementing the programs entirely in hardware. We are evaluating the number of "bit-operations" required, which is roughly the same as the number of Boolean gates. While we can give a relatively reasonable estimate for arithmetic operations, the complex operations (memory access, call, return) require measurements of a real implementation. Moreover, the cost of the complex operations may be dependent on the program size. For example, a program with many procedures may require a complex dynamically routed communication network for handling procedure invocations. While a program with just a few may have direct caller/callee connections. Figure 3.24 shows the hardware resources for each of the programs, using the arguably volatile metric of resources per lines of source code.

We can nevertheless use these measurements to get a coarse estimate of the capabilities of ASH. For example, assuming that a complex operation requires about 200 bit-operations, and using the geometric average from Figure 3.24, we deduce that we can expect to use approximately $50 \times 100 + 25 \times 200 = 10000$ bit-operations for every 100 lines of code. On a hardware budget of 1 million bit operations, we can expect to be able to implement programs as large as 10,000 lines of code completely in hardware. In Section 7.5 we present actual resource measurements on synthesizable Verilog generated from CASH's back-end.

Chapter 4

Program Optimizations

4.1 Introduction

In this chapter we discuss in some detail the optimizations performed by CASH. We make several research contributions to the subject of compiler optimizations:

- We present several new, more powerful versions of classical optimizations. Some of them are applicable in the context of traditional compilers as well, while others are more intimately tied to the features of the Pegasus internal representation.

- We suggest a new class of optimizations, based on *dynamic dataflow analysis*. As an example we present in Section 4.6 a new, optimal algorithm for register promotion which handles inter-iteration promotion in the presence of control-flow. The idea of these optimizations is, instead of using static conservative approximations to dataflow facts, to *instantiate the dataflow computation at run-time* and therefore to obtain dynamic optimality.

- We show that Pegasus can uniformly handle not only scalar optimizations, but also optimizations involving memory. Other efficient program representations treat memory as a second-order object, resorting to external annotations and analyses in order to handle efficiently side-effect analysis. For example, while SSA [CFR$^+$91] is an IR that is widely recognized as enabling many efficient and powerful optimizations, it cannot be easily applied in the presence of pointers. (Several schemes have been proposed [LCK$^+$98, SJ98, BGS00, CG93, LH98, CLL$^+$96, Ste95], but we believe that none of them fits as naturally as Pegasus in the framework of single-assignment.) We show how Pegasus enables efficient and powerful optimizations in the presence of pointers.

- We show how Pegasus not only handles optimizations naturally in the presence of predication and speculation (most classical optimizations and dataflow analyses breakdown in the presence of predication), but even takes advantage of predicates in order to implement very elegant and natural program transformations without giving up any of the desirable features of SSA. We show how to use predication and SSA together for memory code hoisting (subsuming partial redundancy elimination and global common-subexpression elimination for memory operation), removing redundant loads, loads after stores, and dead stores in Section 4.4.

- Unlike other intermediate representations, Pegasus represents essential information in a program without the need for external annotations. Moreover, the fact that Pegasus directly connects definers and users of values makes many traditional dataflow analyses unnecessary: they are summarized by the edges. Therefore, many optimizations are reduced to a local rewriting of the graph. This is important not because of the elimination of the initial dataflow computation, but because *dataflow information is maintained automatically in the presence of program transformations*. Therefore, there is no need for cumbersome (incremental or complete) re-computations of dataflow facts after optimizations.

- As a by-product of the sparsity of the representation and of the lack of need of dataflow analyses, many optimizations are linear time, either worst-case or in practice. This enabled us to use a very aggressive optimization schedule by iterating most optimizations until convergence, shown in Section 4.2. We believe that CASH is unique in this respect. All compilers that we know of have a relatively fixed schedule of optimizations, while in CASH some optimizations may be applied many times.

- The sparsity of Pegasus and its "def-use" nature makes the implementation of dataflow analyses practically trivial. We illustrate this by describing a general dataflow framework and showing how several analyses fit into it in Section 4.7.

- In Section 4.5.4 we introduce *loop decoupling*, a new loop parallelization algorithm for handling memory dependences which relies on creating a computation structure that *dynamically controls the slip* between two parallel computations.

- We bring new insights into the implementation of software pipelining. Our approach, inspired by the paradigm of dataflow software pipelining, was devised for implementation in Spatial Computation and is discussed more fully in Section 6.7. Our approach is immediately applicable on multithreaded processors and is much simpler than traditional software pipelining, while being also tolerant of unpredictable latency events.

An important caveat is in order: many of the optimizations we describe operate naturally (only) at the hyperblock level. Therefore, while they are stronger than basic-block optimizations, they are weaker than global (whole procedure) optimizations. The price we pay for speed and compactness is some level of sub-optimality. While it is relatively difficult to characterize the loss, we believe that the benefits we get in terms of simplicity and speed outweigh the loss in global optimality.

Let us note an interesting phenomenon: while the CASH compiler was devised as an exploration into the subject of application-specific hardware, perhaps the most important by-product of this research has been the creation of several new compiler algorithms, most of which are applicable in traditional settings. However, we believe that the discovery of these optimizations has been substantially facilitated by the *change in perspective* brought by this new viewpoint of compilation, seen as synthesis of Spatial Computation. These results show again that research is an unpredictable journey and that setting remote goals can lead one on fruitful paths, even if they do not always end at the expected destination.

For some optimizations Pegasus allows extremely efficient implementations, much simpler than when using other IRs. The standard we compare to is always the description of the optimization given in [Muc97]. The actual code implementing these optimizations is extremely compact and easy to understand. Table 4.1 shows the complete amount of C++ code required to implement each optimization, including whitespace and comments.

In this chapter we discuss the various optimizations implemented in Pegasus. Some optimizations are barely mentioned. Most of them are versions of well-known program transformations rewritten to take advantage of Pegasus. Some are completely new. We discuss pragmatic issues even for some of the well-known transformations, highlighting aspects ignored in the literature about how to efficiently handle special

Optimization	**LOC**
Hyperblock optimizations	
Constant folding	750
Algebraic optimizations	950
Strength reduction (constant multiply, divide, modulo)	502
Mux simplifications	255
Re-association	220
Loop-invariant code motion	195
Induction variable strength reduction	118
Transitive reduction of tokens	187
Common subexpression elimination	282
Induction variable discovery	646
PRE for memory	235
Inter-iteration register promotion	309
Lifting loop-invariant memory accesses	284
Loop decoupling	180
Memory disambiguation	257
`independent` pragma disambiguation	293
Reachability test	127
Global optimizations	
Unreachable hyperblock removal	38
Dead local variable removal	99
Global dead code elimination	41
Global constant propagation	99

Table 4.1: *C++ lines of code (including whitespace and comments) required for implementing major Pegasus optimizations.*

cases or how to engineer an efficient implementation. The hope is that these highlights may be useful to other compiler developers, even if their value as a research contribution is hard to quantify.

4.2 Optimization Order

An important subject in current compiler research is the actual *ordering* of the optimizations (e.g., [TVVA03, YLR$^+$03]). While we cannot claim to solve this problem, we believe that the approach taken in the CASH compiler is an interesting and rather extreme design point in the space of possibilities.

It is well known that some optimizations have to be repeatedly applied in order to be effective or to speed-up the compilation process [Muc97]. CASH actually iterates many optimizations until no changes are observed in the graph. Some optimizations may be therefore applied more than 10 times over each program fragment![1] This should make the final result somewhat less sensitive to optimization order (although certainly not completely insensitive). Most optimizations are amortized linear-time because they are applied separately to acyclic code regions (i.e., individual hyperblocks) and therefore require a single pass.

[1] We have instrumented the compiler to provide warnings if more than 10 different iterations are required to convergence. With the current optimization order and input programs this warning was never displayed.

Another important aspect of optimization efficiency is that Pegasus already encodes explicitly many important dataflow facts, such as liveness, reachability, def-use, and may-dependence. Therefore, optimizations do not need to compute and more importantly, recompute these dataflow facts.

The current ordering for the optimizations has been achieved mostly by experimentation. Some of the choices can be justified on relatively solid grounds, some were observed to produce better results, while some were found not to have too much of an importance. We have unfortunately noticed that as more optimizations passes are added, the results seem to become *more* sensitive to the ordering.

The optimization process is broken into two parts — core optimizations and post-processing optimizations:

Core optimizations: these consist of a series of code transformations, most of which are applied repeatedly. The bulk of the optimization process occurs in this phase. The pseudo-code for this phase is given in Figure 4.1. However, some optimizations in this phase are only applied once, either because they are idempotent, or because applying them repeatedly can lead to code degradation.

Post-processing optimizations: these are a series of code transformations applied only once on the program in a very careful order. Figure 4.2 shows the pseudo-code for this optimization phase. All post-processing optimizations are target-oriented (i.e., they are motivated by the hardware target). There are two types of post-processing optimizations:

(a) Optimizations that break some representation invariants. For example, a “mu” node with a single input is removed. Such a removal cannot occur earlier, since most code analysis expects each hyperblock to start with “mu” nodes (except the entry hyperblock, which starts with “argument” nodes).

(b) “Reversible” term-rewriting optimizations that break a canonical representation of a formula. An example of such an optimization is rewriting `a + (-b)` as `a - b`. Section 4.3.2.1 gives a complete list and a more detailed explanation.

Most of the non-obvious optimizations are described in detail in the rest of this chapter.

4.3 Classical Optimizations

Pegasus’ strength is apparent when we examine how it expresses program optimizations. In this section we examine informally most of the scalar optimizations described by Muchnick [Muc97] and their implementation in Pegasus. The next sections are devoted to novel optimizations.

Before reviewing the optimizations, it is important to note that Pegasus converts all control flow into data flow, so some optimizations no longer strictly correspond to their original definition. For example, straightening is an operation that chains multiple basic blocks in a single basic block under some appropriate conditions. However, there are no basic blocks in Pegasus, so we cannot speak of straightening any longer. Nevertheless, we define a transformation $\mathcal{S}$ on Pegasus and claim it is equivalent to straightening, in the following sense: if we apply straightening on a CFG and next translate to Pegasus, we obtain the same result as if we translate the CFG to Pegasus and next apply $\mathcal{S}$. In other words, the following diagram commutes:

$$\begin{tikzcd}
\text{CFG} \arrow[r] \arrow[d, "\text{straightening}"'] & \text{Pegasus} \arrow[d, "\mathcal{S}"] \\
\text{optimized CFG} \arrow[r] & \text{optimized Pegasus}
\end{tikzcd}$$

```
global_optimizations() {
    do {
        unreachable_hyperblock_removal();
        global_constant_propagation();
        global_dead_code();
    } while (changes);
}

hyperblock_optimizations(hyperblock h) {
    reassociation(h);
    do {
        transitive_token_reduction(h);
        cse(h);
        term_rewriting(h);
        pre_for_memory(h);
    } while (changes);
    boolean_optimizations(h);
}

core_optimizations() {
    repeat twice {
        for each hyperblock h
            hyperblock_optimizations(h);
        global_optimizations();
    }

    /* One-time optimizations */
    for each hyperblock h {
        reassociation(h);
        hyperblock_optimizations(h);
        immutable_loads(h);
        memory_disambiguation(h);
        loop_invariant_code_motion(h);
        loop_register_promotion(h);
        loop_decoupling(h);
        loop_strength_reduction(h);
    }
    remove_unused_locals();
    build_local_stack_frame();

    repeat twice {
        for each hyperblock h
            hyperblock_optimizations(h);
        global_optimizations();
    }
}
```

Figure 4.1: *Pseudo-code for the order of the core optimizations in CASH. Many of these passes also invoke local (hyperblock-level) dead-code elimination as a clean-up phase.*

```
post_processing() {
    for each hyperblock h {
        if (no_computation(h))
            remove(h)
        else
            create_switch_nodes(h);
    }

    for each hyperblock h {
        term_rewriting(h);
        reversible_optimizations(h);
        pipeline_balancing(h);
        pipeline_operations(h);
    }

    global_dead_code();
}
```

Figure 4.2: *Post-processing optimizations in CASH.*

4.3.1 Free Optimizations

These optimizations are implicitly executed during the creation of the Pegasus representation. They do not require any code to be implemented.

Copy propagation and constant propagation at a hyperblock level occur automatically during the construction of the SSA form.

Straightening and branch chaining occur because unconditional branches are transformed into constant "true" predicates that are removed from all path expressions by Boolean formula simplifications.

Unreachable code elimination occurs during the partition in hyperblocks because the depth-first traversal does not assign unreachable blocks to any hyperblock.

Removal of `if` statements with empty branches occurs due to predicate simplification and dead-code elimination. Basic blocks before and after an empty conditional are control-equivalent, therefore the conditional expression from an empty `if` will not control any multiplexor and in consequence becomes dead code.

4.3.2 Term-Rewriting Optimizations

All these optimizations look for a pattern subgraph in the Pegasus graph and then replace it with another pattern with equivalent functionality. Most patterns are constant-size, while some others can be arbitrarily large. A pattern is matched always starting from the "root", by searching its descendants. Figure 4.3 illustrates schematically the pattern-replacement process. The replacing pattern is synthesized separately and the outputs of the replaced graph are rewired from the replacement. The old pattern is left in place to

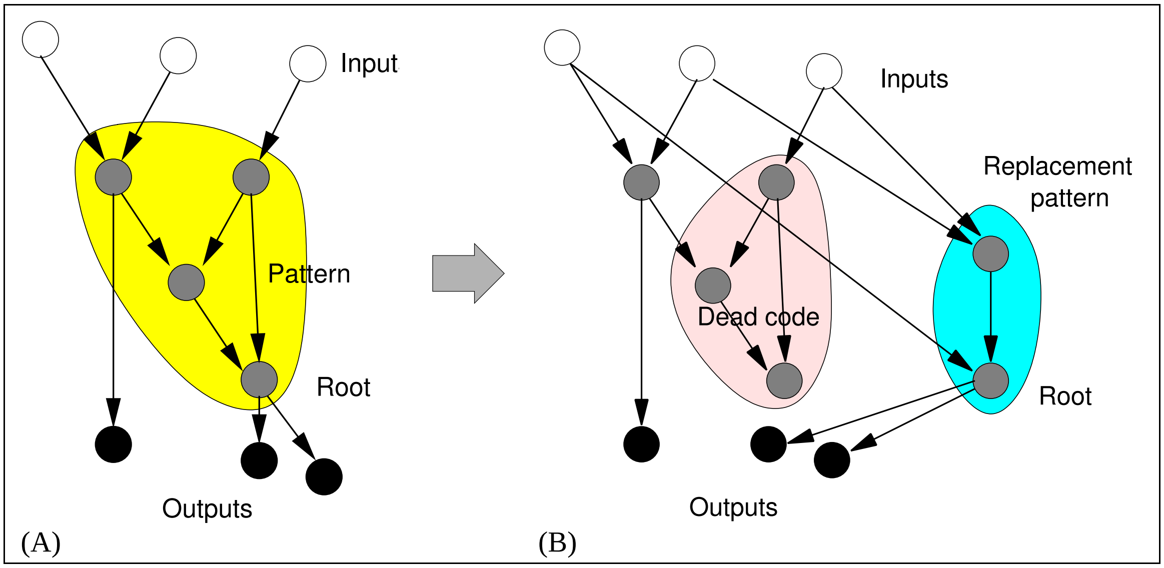


Figure 4.3: *Generic term-rewriting optimization: a pattern is always a DAG matched starting from the root and replaced with a different DAG.*

be cleaned up by dead-code elimination. This approach is particularly convenient if the DAG has multiple "outputs" and not all of its nodes become dead.

Term-rewriting optimizations include new algebraic optimizations involving the new constructs (multiplexor, mu, eta) as shown in Figure 4.4 and Figure 4.5(c). A special form of dead-code can be applied to predicated operations guarded by constant "false" predicates, as illustrated for some examples in Figure 4.5. Such operations can be simply discarded, since their result is irrelevant (a "false" predicate indicates that control-flow can never reach such an operation, or that the operation is/has become redundant as shown in Section 4.4).

Collectively *all* the term-rewriting optimizations can be applied in linear time to a hyperblock, by processing the graph in topological order (ignoring the back-edges). The pseudo-code is given in Figure 4.3. This code is linear time under the assumption that all source and replacement patterns have constant size.

4.3.2.1 Reversible Optimizations

In some cases there are reasons for performing a term-rewriting either of pattern A into B or of B into A. For example, $a - b$ for integer types is the same as $a + (-b)$. The former takes fewer operations, while the latter may enable other optimizations, for example re-association, if a is itself the result of an addition. So which way should the compiler perform the transformation?

The answer to this question in CASH is: both. Usually this question will arise if one form of an expression is better at enabling other optimizations, and the other reduces run-time or hardware area. Therefore the solution is to dub the form which enables other optimizations a "canonical" form. This idea has been advanced in [Fra70]. During the bulk of the optimizations, the compiler brings all such expressions to the canonical form. After the main optimizations are completed the compilers enters the "post-processing" phase (see Section 4.2) and at this point each canonical form is dismantled into its corresponding minimal form. Table 4.3 lists all such optimizations currently applied.

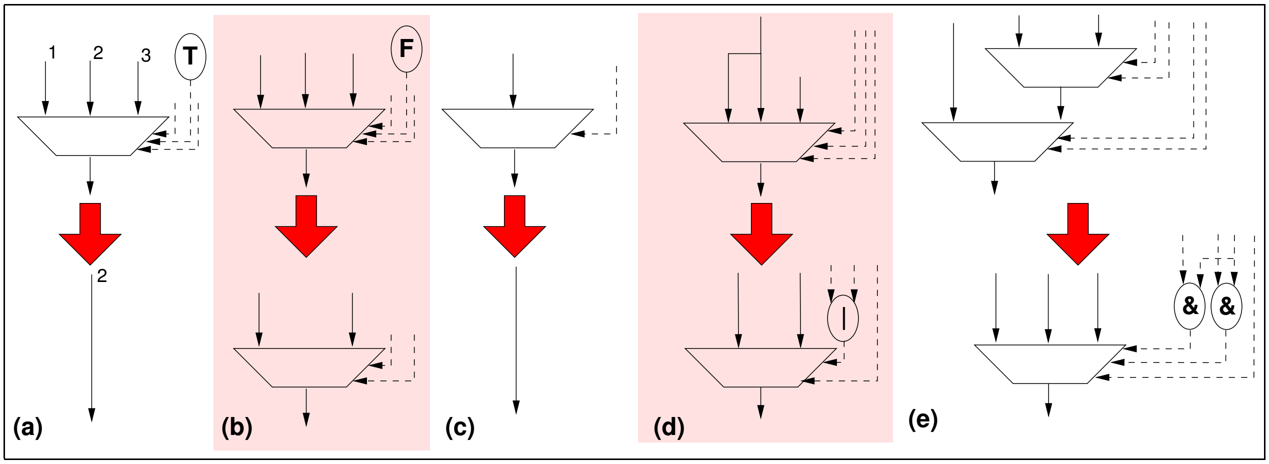


Figure 4.4: *Algebraic simplifications for multiplexors: (a) a multiplexor with constant "true" predicate is replaced with the selected input; (b) multiplexor with constant "false" predicate has the selected input removed; (c) multiplexor with only one input is removed; (d) multiplexor with two identical data inputs merges the inputs and "or"s the predicates; (e) chained multiplexors.*

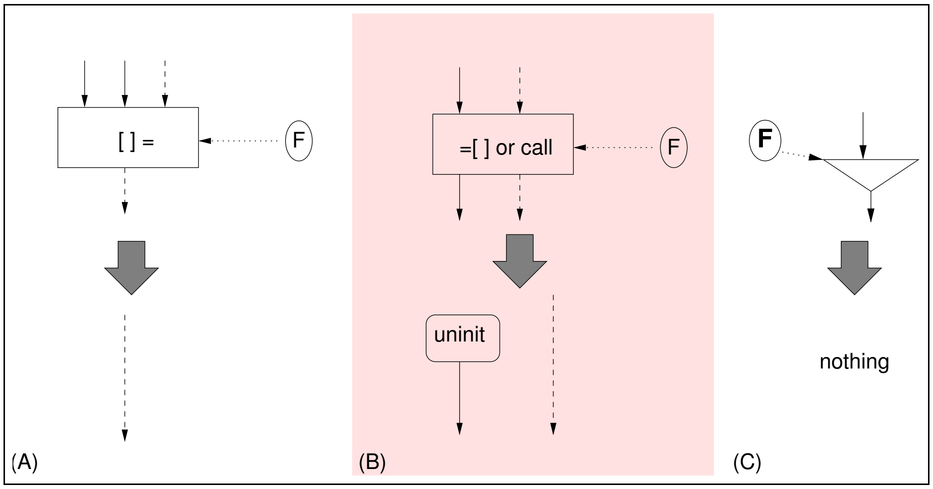


Figure 4.5: *Optimization of operations controlled by false predicates: (A) stores, (B) loads and calls, (C) etas. Since these operations can never be dynamically executed, they can be safely removed from the graph. For (A) and (B), the token edge is "short-circuited" from the input to the output because it must preserve dependences between prior and later operations. The data generated by a removed call or load can be simply replaced with an uninitialized node. The uninitialized node will most likely be proved dead and removed by further optimizations.*

Optimization	**Comments**
Constant folding	Quite complicated to implement due to the large number of combination of possible operations and types. This optimization is also quite hard to implement if the target machine is not the same as the compiler machine (e.g., in cross-compilation). CASH's implementation uses native arithmetic to carry the constant folding.
Strength reduction	The code for constant multiplication, constant division and modulus [GM94] has been adapted from gcc.
Induction variable strength reduction	Multiplications of induction variables with constants are turned into repeated additions.
Algebraic optimizations	`a+0`, `a-0`, `a*0`, `a*1`, `a*-1` as `-a`, `0/a`, `a/1`, `a>>0`, `a<<0`, `a|0`, `a&0`, `˜a-1 as -a`, `0-a`, `-1-a` as `˜a`, `0/a`, `(a+C1)*C2` as `a*C2+C1*C2`, `a%1` as `0`, `0%a`, `-a/-b`, `a%-b as a%b`, `a-b==0` as `a==b`, `(a-b)!=0` as `a!=b`, `-a==C` as `a==-C`, `-a*-b` as `a*b`, `!!a` as `a`, `!(a==b)` as `a!=b`, `a&a`, `a&˜a`, `˜a&˜b` as `˜(a|b)`, `a<<b<<c` as `a<<(b+c)`, `--a`, `-(a-b)` as `b-a`, `a-(-b)` as `a+b`, `a-b` as `a+(-b)`, `a-(a&b)` as `a&˜b`, `a-a`, `a+a` as `a<<1`, `a==a` as `1`, `a!=a` as `0`, `-a<-b` as `a>b`, `a*C+a` as `(a+1)*C`, `(a*b)+(a*c)` as `a*(b+c)`. (C stands for a constant; none of these is applicable to floating-point.)
False predicates	See Figure 4.5.
Multiplexor optimizations	See Figure 4.4.
PRE for memory	See Section 4.4.

Table 4.2: *Term-rewriting optimizations*

Canonical form	**Minimal form**	**Comments**
`˜a`	`-1 - a`	
`a - b == 0`	`a == b`	Also for `!=`
`a + b == 0`	`a == -b`	Also for `!=`
`!(a == b)`	`a != b`	Simpler Boolean tree leaves
`!(a < b)`	`b <= a`	
`a + (-b)`	`a - b`	
`a + a`	`a << 1`	
`(a * b) + (a * c)`	`a * (b+c)`	Also for `-`
`and(a, b, c)`	`and(and(a, b), c)`	Also for `or`, `V`, `mux`

Table 4.3: *Reversible optimizations used in CASH.*

The minimization transformation on the last line in Table 4.3 breaks some associative operations (Boolean and, or, token combines and muxes) with large fan-in into an un-balanced tree of operations with a smaller fan-in. The operation for muxes basically undoes the transformation from Figure 4.4(e). In this example, the minimal form is faster when input `c` is the last one to arrive.

```
sort topologically hyperblock ignoring back-edges

foreach node n in topological order {
    switch (n->operation) {
        case addition:
           if (root_of_pattern_1(n))
               replace(n, replacement_pattern_1);
           else if (root_of_pattern_2(n))
               replace(n, replacement_pattern_2);
           else
               ...
           break;
       case multiplication:
           ....
    }
}
```

Figure 4.6: *Pseudo-code for performing term-rewriting optimizations.*

4.3.3 Other Optimizations

Dead code elimination (at the hyperblock level) is a trivial optimization: each side-effect-free operation whose output is not connected to another operation can be deleted. Even loads whose data output is unconnected can be removed. The removal of a load connects the token input to the output, as for the removal of loads with constant “false” predicates illustrated in Figure 4.5. Once an operation has been removed, the operations producing its inputs are also candidates for removal if their output had a fanout of 1. Dead code removal is therefore a linear-time operation, which is very frequently invoked to perform clean-up after term-rewriting optimizations. As shown in Figure 4.3, term-rewriting replaces the root of a pattern with a different pattern, leaving the old root “hanging” in the circuit. A dead code pass is used to clean-up after a batch of term-rewriting operations has been performed in topological order.[2] Performing dead-code elimination on each hyperblock separately repeatedly brings most of the benefits, but does not guarantee the removal of all dead code. A complete solution, relying on a dataflow analysis, is described in Section 4.7.2.

Re-association is the process of reordering a chain of commutative and associative operations. We used the algorithm from [BC94]. Trees of associative and commutative operations are re-ordered to operate first on constants, then on link-time constants, then on loop-invariant values and then on all other terms. This ordering enables constant propagation and loop-invariant code motion for the corresponding parts of the tree. The trees are also height-balanced during this process. An important question is the re-association of DAGs: how should nodes with fanout be handled? Our current implementation blows up these nodes for each output, potentially creating an exponentially-sized tree out of a DAG. While in practice this has not been a problem, a more careful implementation ought to test how large the expansion is and stop duplicating subtrees if the cost is excessive.

[2] Doing term-rewriting by checking roots in topological order ensures that newly created dead code is never considered for term-rewriting.

Constant conditionals in `if` statements are eliminated by algebraic simplifications for multiplexors and etas. Such constant conditionals are translated into constant selectors for multiplexors or etas, which can be simplified as in Figure 4.4(a,b) and Figure 4.5(c).

More unreachable code removal, for code proved unreachable due to constant propagation, is achieved by removing the hyperblocks having *merge* nodes with no inputs.

SSA form minimization is obtained after repeatedly simplifying multiplexors as shown in Figure 4.4.[3]

Global common-subexpression elimination (CSE), value numbering, partial redundancy elimination are actually all the same operation, due to the speculation and copy-propagation. The algorithm to implement all of these is trivial and is as complex as strictly local traditional common-subexpression elimination. A prerequisite for an efficient CSE is that constant nodes are unique per hyperblock. Then all nodes performing the same computation and having exactly the same inputs can be "coalesced." A linear time implementation can be obtained by hashing nodes on their inputs. To handle commutative operations, the inputs of each operation are first sorted in an arbitrary canonical order.[4]

Even operations with side-effects (memory loads and stores, function calls and returns) that have identical inputs (including the predicate) are coalesced by CSE. CSE can be performed truly globally: nothing prevents us from applying CSE to two mu nodes having the same inputs coming from a different hyperblock.

Note that since side-effect-free operations had their predicates promoted to "true", CSE may be applied to operations residing originally in different basic blocks. This is why CSE in Pegasus is as powerful as global value numbering.

Scalar promotion using a new optimal algorithm which is extensively described in Section 4.6.

Code hoisting and tail merging are subsumed by GCSE because expressions which compute the same value in different basic blocks of the same hyperblock can be merged by CSE after predicate promotion.

Loop-invariant code motion is illustrated in Figure 4.7. A loop-invariant expression is defined [Muc97] as (1) a constant; (2) an expression not changed within the loop; or (3) a computation having all inputs loop-invariant. In this figure, (1) the 5 is a constant; (2) the mu-eta loop carries an unchanged value; and (3) the addition has all inputs loop-invariant.

Finding the maximal loop-invariant code then amounts to a simple hyperblock-level dataflow computation in two steps: (a) marking all constant and simple loop-invariant expressions (i.e., just a trivial mu-eta cycle, with no intervening computation; see also induction variable discovery in Section 4.3.4) and (b) traversing the graph forward and marking the expressions reachable only from already marked expressions. Lifting a loop-invariant expression "before" the loop requires the creation of a new *mu–eta* cycle, which is used to carry around the newly created loop-invariant value (the leftmost loop in Figure 4.7(B)).

In order to lift loop-invariant expressions out of a loop a *loop pre-header* hyperblock is created. Since each loop has a single entry point, it is relatively easy to insert a new hyperblock before the loop: all mu input edges coming from other hyperblocks are diverted to the pre-header. The pre-header branches unconditionally to the loop: all its eta nodes are controlled by a constant "true."

[3] A minimal SSA representation results for each individual hyperblock. The form is not globally minimal.

[4] Strictly speaking sorting is not linear time, since some operations, such as Booleans, can have arbitrarily large fan-in.

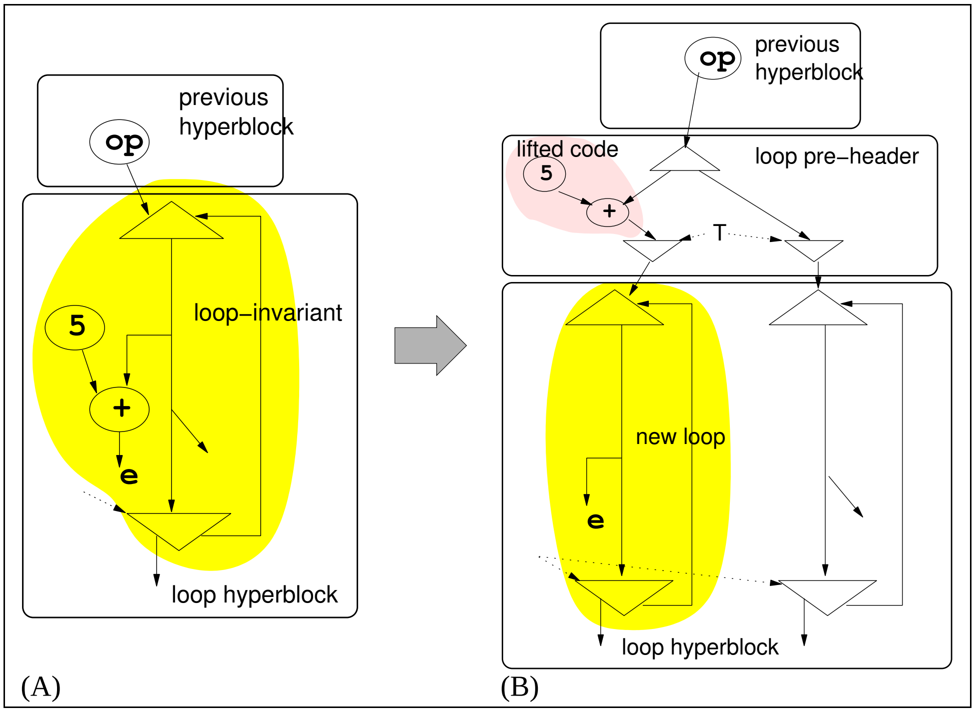


Figure 4.7: *Loop-invariant code motion.*

4.3.4 Induction Variable Analysis

Induction variable analysis has the purpose of discovering expressions whose value grows by a constant amount in every iteration of a loop. They include loop-invariant expressions (whose value grows by 0). Induction variable analysis proves to be extremely powerful since (a) it can be applied to very low-level code, such as Pegasus, and (2) the information it produces can be used for a variety of complex optimizations.

For induction variable discovery we use the excellent linear-time algorithms proposed by Wolfe [Wol92], which are described on an SSA form. The discovery is a three step-process: (1) find loop-invariant values (0-th level induction variables); (2) find basic induction variables: strongly connected components that amount to a change by a constant step in every iteration; (3) finding derived induction variables which are linear combinations of basic induction variables.

Information about induction variables is stored in separate program annotations. This is the first example of an important program summarization that is not directly represented in Pegasus. Since induction variable annotations are “external” objects, they are not automatically updated on program transformations. However, since computing this information is a linear-time process, the information is simply discarded and recomputed from scratch every time it is needed.

Unfortunately, the current representation of induction variable information is not as general as it could be. The information is attached to a node and describes the behavior of the node with respect to successive iterations. The attached information contains two fields: a base, which is the initial value of the node output; and a step, which indicates the change in every iteration. The base is represented as a *linear combination* of values of other nodes. Although the step can theoretically be any loop-invariant expression, the current representation only handles compile-time constant values.

Figure 4.9 shows an example labeling of induction variables for a small program. Nodes [12],[51],[77]

```
void f(void)
{
  extern int xx[];
  int i;

  for (i=1; i<10; i++)
  {
    xx[i]++;
    if (i != 5)
      xx[i+2] = xx[i];
  }
}
```

Figure 4.8: *Sample program for illustrating induction variable analysis.*

contribute to the computation of the basic induction variable `i`. Their step is 1. The base value for node [12] is 1, obtained from scanning the predecessor hyperblock. Node [134] is a loop-invariant value, actually a link-time constant, with base `xx+8` and step 0.[5] Finally, nodes [108] and [136] are derived induction variables. Node [136] has the most complicated expression: step 4 and base `xx`+12 (i.e., `xx`+4*[12]+8).

One of the most useful applications of induction variable information in CASH is for analyzing memory accesses. In this example it is clear that the load [24] and the store [29] access the same memory locations, since they have the same address, the output of [22]. Comparing the addresses of the load [24] and store [49] is easily achieved: since they have the same step 4, it is enough to subtract their bases: (`xx`+12) – (`xx`+4) = 8. Since the difference is greater than the size of the data accessed (both operate on 4 bytes), this enables CASH to conclude that:

- Accesses [24] and [49] are always independent, therefore there is no need for a token edge between them.

- They have a constant dependence distance of $(base_1 - base_2)/step = 8/4 = 2$ (i.e., operation [49] accesses the same locations as [24] two loop iterations later). This information is used for register promotion (Section 4.6).

Notice that induction variable analysis has enabled the extraction of useful information from a rather low-level program representation. For example, no information is used about the fact that [24] and [49] correspond to `x[i]` and `x[i+2]` respectively. Therefore, this analysis would work even if the accesses were written with pointer accesses (`xx+2`) and not with arrays.

Our initial implementation of induction variable analysis was applied on a per-hyperblock basis. However, this led to an interesting problem: loop-invariant code lifting led to a decrease of the quality of the induction variable analysis. Consider Figure 4.7 again, where `e` is loop-invariant. In the left-hand side of this figure, CASH can easily prove that `e = op+5`. However, in the right-hand side the computation `op+5` has been lifted out and replaced with a different mu-eta loop. Just from the loop hyperblock CASH cannot

[5] In standard C notation, `xx` stands for the address of the first element of the array `xx`. Pointer arithmetic is always made on bytes, using the Suif convention, and since an integer has 4 bytes, the `&xx[2]` is `xx+8`.

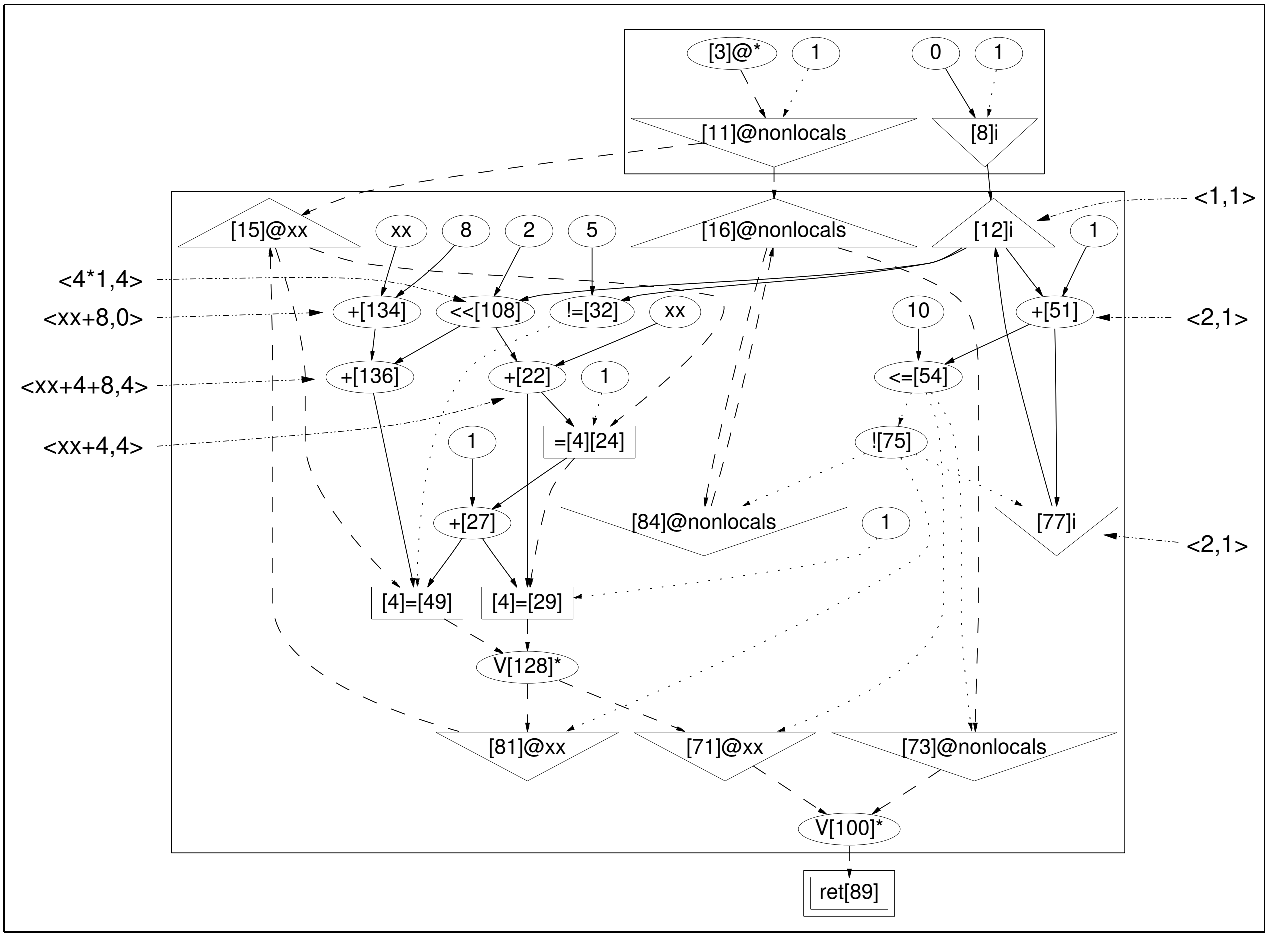


Figure 4.9: *Pegasus representation of the program in Figure 4.8 with labeled induction variables. The labeling has the form* <base,step>*. The* base *is a linear combination of constant values and other node values. Nodes numbers are within square braces. Node [134] is the value of* `&xx[2]`*, node [136] the value of* `&xx[i+2]`*, and node [22] is* `&x[i]`*.*

prove this relationship anymore. Therefore we had to extend the algorithm for the computation of the base to also analyze the predecessor hyperblock of the loop, if it is unique.

Induction variable analysis could still be improved in CASH to handle arbitrarily loop-invariant steps and not only compile-time constants. This kind of information could potentially enable further optimizations.

4.3.5 Boolean Minimization

As we show in Section 4.4, CASH relies extensively on predicate manipulation for implementing optimizations. In CASH predicate manipulation is the counterpart of CFG restructuring in traditional compilers: since the predicate guarding an operation corresponds uniquely to a basic block, modifying this predicate corresponds to moving the instruction to a different basic block.

Predicate computation is also very important for performance, as shown in Chapter 8, since predicates control the eta operations which feed data to the successor hyperblocks. These predicates correspond directly with CFG branch conditions. Therefore, efficient predicate computations are paramount for effective compilation.

Unfortunately predicate expressions can become quite involved due to the unrolling of loops with statically unknown bounds and due to the memory optimizations from Section 4.4. As discovered also by [SAmWH00], predication can build Boolean expressions with hundreds of variables.

There are two types of beneficial predicate simplifications: first, proving that some predicates are constant may trigger other optimizations. Proving predicates "false" may enable removal of controlled operations. Proving predicates "true" may enable loop-invariant lifting of controlled operations and better register promotion. Second, reducing the depth of a Boolean formula may speed-up its evaluation at run-time.

For the purpose of Boolean simplification CASH uses three different methods:

- For Boolean networks with few inputs (i.e., less than 16), CASH performs an exhaustive evaluation of the Boolean function and discovers constant-valued nodes.

- For testing conservatively certain Boolean relationships between predicates (e.g., $p \Rightarrow q$), CASH uses simple pattern-matching (e.g., $q = p \wedge r$, for some r). It is important to note that these tests are not exact and that they may err always conservatively (e.g., in this case the test may decide that $p \not\Rightarrow r$ even if $p \Rightarrow r$, but it will never decree that $p \Rightarrow r$ when it is not true).

- Finally, to reduce the complexity of Boolean expressions, CASH uses the Espresso CAD tool from Berkeley [BSVHM84]. Espresso rewrites Boolean formulas given as a tabular form using a variety of heuristics and outputs a sum-of-products form. If a Boolean network is too large to write as a table, CASH tries to optimize separately each output of the network. Even so some formulas are too large and therefore may be unoptimized.

Neither of these three methods is ideal. In the future we plan to transition to a different set of tools:

- The integration of a BDD [Bry86]-based package is under way [Som]. BDDs in general will allow for a more complexity-effective testing for constant values and for exact Boolean tests of relationships (e.g., $p \Rightarrow q$).

- For depth reduction we plan to transition to a more appropriate tool such as SIS [SSL$^+$92], which does not require the input to be in two-level form. A better tool will enable us to handle Boolean networks with a larger number of variables.

- Finally, currently Boolean optimization treats the leaves of the Boolean network as opaque. For example, for integer variables a and b, the Boolean expression $(a < b) \vee (a >= b)$ treats the two comparisons as independent variables. However, in this case the two variables are clearly complementary and the Boolean expression can be reduced to "true." This kind of information can be integrated in BDDs using techniques described in [SAmWH00, ASP$^+$99], and in Boolean minimizers using don't cares.

Predicate-carrying eta operations can be sometimes specially optimized: let us consider an eta node with predicate p and whose data value is a predicate as well, d. Then, if $p \Rightarrow d$, one can safely replace d with "true." If $p \Rightarrow \neg d$, then d can be replaced with "false." These optimizations are quite important because they may create new opportunities for global constant propagation of d.

```
void f(unsigned*p,
       unsigned a[],
       int i)
{
    if (p)
        a[i] += *p;
    else
        a[i] = 1;
    a[i] <<= a[i+1];
}
```

Figure 4.10: *Simple code example manipulating memory. Optimization of this example is shown in Figure 4.11. No assumption is made on the aliasing of* `p` *and* `a`*.*

4.4 Partial Redundancy Elimination for Memory

The previous sections of this chapter have been devoted to the discussion of scalar optimizations in Pegasus. We have seen that most of these can be handled very efficiently in a quasi-global form (e.g., per-hyperblock) due to the SSA representation. This section shows that optimizing memory accesses, a class of optimizations that require substantial machinery in other IRs — be they based on CFGs or SSA — can be tackled cleanly when using Pegasus. The core assumption behind all the memory optimizations is that it is legal to temporarily keep memory values in registers, perform updates on these registers and only store the result at the end (i.e., use register promotion). The semantics of C permits such manipulations. An early version of this section has been published as [BG03b].

As described in Section 3.2 and Section 3.3, Pegasus uses *token edges* to indicate may-dependences between memory operations. We show here how this representation can be seen as a kind of SSA for memory itself, and how its sparsity enables very efficient dependence tests. We also show a novel way of performing register promotion and partial redundancy elimination for memory by manipulating strictly the predicates associated to operations, without any CFG-like code restructuring.

4.4.1 An Example

We illustrate succinctly the power of the Pegasus framework for handling memory with the example in Figure 4.10. This program contains no loops; it uses `a[i]` as a temporary variable. A good optimizer for memory accesses ought to use a scalar register instead of going to memory for the temporary.

We have compiled this program using the highest optimization level using eight different compilers, shown in Table 4.4. Only CASH and the AIX compiler removed all the useless memory accesses made for the intermediate result stored in `a[i]` (two stores and one load). The other compilers retain the intermediate redundant stores to `a[i]`, which are immediately overwritten by the last store in the function. We believe that, since none of the pointers is declared `volatile`, they do so not in order to preserve exceptions but because this optimization is not fully implemented. CASH performs very simple analyses and transformations for achieving this goal: only reachability computations in a DAG and term-rewriting code transformations. The point of this comparison is to point out that optimizations which are complicated enough to perform in a traditional representation (so that many commercial compilers forgo them), are very

Compiler	Architecture	Flags
gcc 3.2	Intel P4	-O3
Sun WorkShop 6 update 2	Sparc	-xO5
DEC CC V5.6-075	Alpha	-O4
MIPSpro Compiler Version 7.3.1.2m	SGI	-O4
SGI ORC version 1.0.0	Itanium	-O4
Microsoft Visual C++ version 12.00.8168	Windows	/Ox
IBM AIX cc version 3.6.6.0	Power4	-O3
CASH	ASH	all

Table 4.4: *Eight compilers used to optimize the example in Figure 4.11*

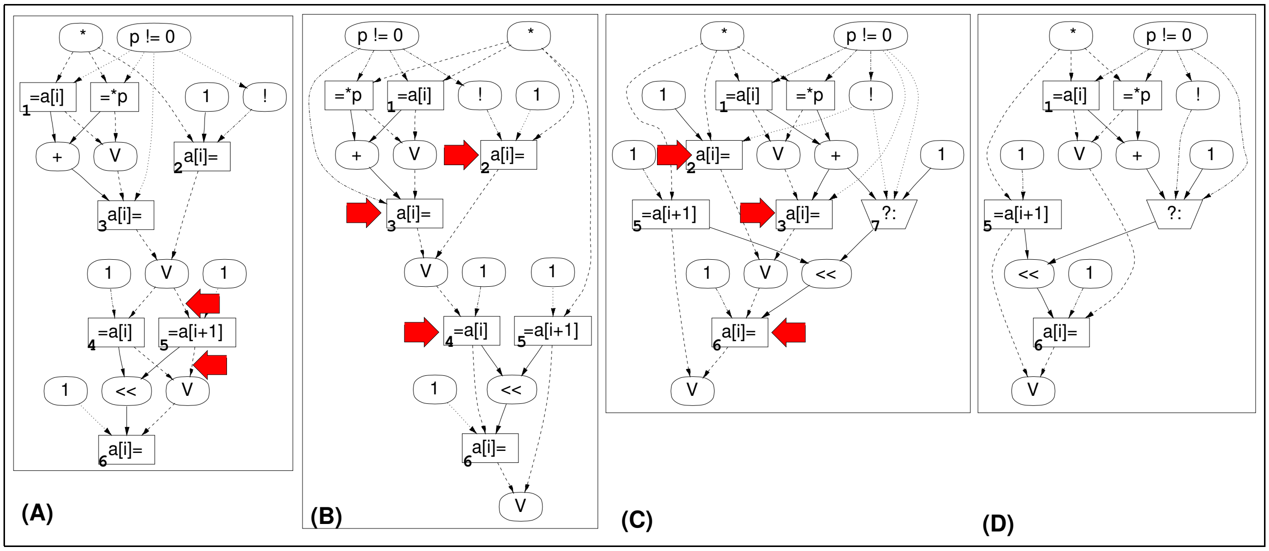


Figure 4.11: *Program transformations for the code in Figure 4.10: (A) removal of two extraneous dependences between a[i] and a[i+1]; (B) load from a[i] bypasses directly stored value(s); (C) store to a[i] kills prior stores to the same address; (D) final result.*

easy to express, implement, and reason about when using a better program representation. The optimization in Pegasus occurs in three steps detailed in Figure 4.11.

Figure 4.11(A) depicts a slightly simplified representation of the program, (we have elided the address computation representation), before any optimizations. Recall that dashed lines represent token edges, indicating *may dependences* for memory access operations. At the beginning of the compilation (Figure 4.10(A)), the token dependences track the original program control-flow.

CASH first proves that `a[i]` and `a[i+1]` access disjoint memory regions and therefore commute (node pairs 3/5, 2/5 and 5/6). By removing the token edges between these memory operations the program is transformed as in Figure 4.11(B). Notice that a new combine operator is inserted at the end of the program. Its execution indicates that all prior program side-effects have occurred.

In Figure 4.11(B), the load from `a[i]` labeled 4 immediately follows the two possible stores (2 and 3). No complicated dataflow or control-flow analysis is required to deduce this fact: it is indicated by the tokens flowing directly from the stores to the load. As such, the load can be removed and replaced with the value

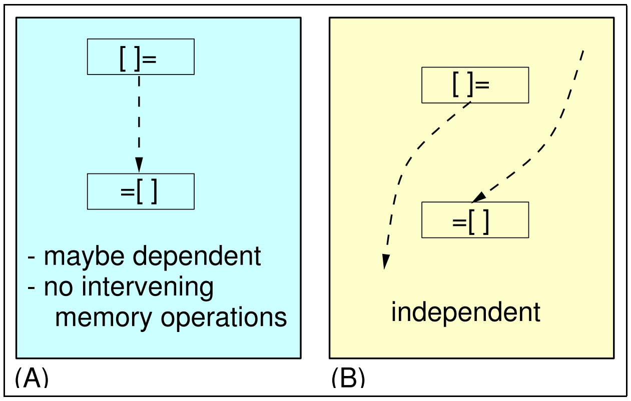


Figure 4.12: *In a transitively reduced graph (A) the presence of a token edge between two memory operations implies the following properties: (1) the two memory operations may access the same memory location,* and *(2) there is no intervening memory operation affecting that location. (B) The absence of a token path indicates complete independence of the operations.*

of the executed store. This replacement is shown in Figure 4.11(C), where the load has been replaced by a *multiplexor* 7. The multiplexor is controlled by the predicates of the two stores to `a[i]`, 2 and 3: the store that executes (i.e., has a "true" predicate at run-time), is the one that will forward the stored value through the multiplexor.

As a result of the load elimination, in Figure 4.11(C) the store 6 to `a[i]` immediately follows (and post-dominates) the other two stores, 2 and 3, to the same address. In consequence, 2 and 3 are useless and can be completely removed, as dead code. Again, the post-dominance test does not involve any control-flow analysis: it follows immediately from the representation because (1) the stores immediately follow each other, as indicated by the tokens connecting them; (2) they occur at the same address; and (3) each predicate of an earlier store implies the predicate of the latter one, which is constant "true" (shown as 1), indicating the post-dominance. The post-dominance is determined by only elementary Boolean manipulations. The transformation from Figure 4.11(C) to (D) is accomplished in two steps: (a) the predicates of the prior stores are "and"-ed with the negation of the predicate of the latter store (i.e., the prior stores should occur only if the latter one does not overwrite them) and (b) stores with a constant "false" predicate are completely removed from the program as dead code.

4.4.2 Increasing Memory Parallelism

CASH performs a series of optimizations with the goal of increasing the available memory parallelism. The basic idea is to remove token edges between memory operations that are actually independent, enabling them to be executed in parallel.

4.4.2.1 Transitive Reduction of Token Edges

As we mentioned in Section 3.3.9.2, the token edge subgraph is maintained in a transitively reduced form during compilation.[6] This invariant is required for the correctness of some optimizations; see Figure 4.12.

[6] That is, if there are edges $a \rightarrow b \rightarrow c$, there will be no edge $a \rightarrow c$.

4.4.2.2 Removing Dead Memory Operations

As mentioned, memory operations are predicated and should be executed only when the run-time value of the predicate is "true." In particular, a side-effect operation having a constant "false" predicate can be completely removed by the compiler from the program. Its token input is connected to its token output (see Figure 4.5). Such constant false predicates can arise due to control-flow optimizations (such as `if` statements with constant tests) or due to redundancy removal, as described in Section 4.4.3.

4.4.2.3 Immutable Objects

Accesses made through pointers to immutable constants (such as constant strings or pointers declared `const`) can be optimized: if the pointed value is statically known, the pointer access can be removed completely and the loaded value can be replaced with the actual pointed value;[7] otherwise, the access does not require *any* token since it does not need to be serialized with any other operation. The access also does not need to generate a token.

4.4.2.4 Memory Disambiguation using Token Edges

One of the important roles of the token edges is to indicate which memory operations have to be tested for dependence by a memory disambiguation pass. Static memory disambiguation is a compile-time technique which determines whether or not two memory instructions (may) access the same memory location at run-time. CASH features a *modular memory disambiguator*, which clearly separates the policy from the mechanism. In fact, it has a plug-in nature for handling multiple policies.

The memory disambiguator chooses pairs of nodes denoting memory accesses and then invokes in order a series of plug-in policies to determine whether the two nodes are dependent. Pseudo-code for the disambiguation engine is given in Figure 4.13.

CASH uses five different policies, roughly in increasing order by cost. What is important is that queries for the disambiguator are only made for the program token edges. Token edges constitute a sparse representation of the essential program dependences. If two nodes a and b are not directly linked by a token edge, then for most purposes it is useless to query the disambiguation engine whether they commute: (1) either there is no token edge path $a \rightarrow^* b$, and then it is known that they do commute, or (2) there is a path $a \rightarrow c \rightarrow^* b$, and then a and b cannot be reordered around c, and therefore it is pointless to check whether they commute.

When two nodes are proved independent, the token edge connecting them is removed. The removal is not trivial: since the transitive closure of the token graph (minus the removed edge) must be preserved, new token edges are inserted. The basic algorithm step is illustrated in Figure 4.14. (Notice that, although the number of explicit synchronization tokens may increase, the total number of synchronized pairs in the transitive closure of the token graph is always reduced as a result of token removal.)

4.4.3 Partial Redundancy Elimination for Memory

In this section we describe several optimizations that take advantage of the SSA-like nature of Pegasus for memory dependences in order to discover useless memory accesses. The explicit token representation makes the following optimizations easy to describe and implement for memory operations: redundant

[7] The access must be a read. A write to a value declared immutable is a type error.

```
enum dependent {
  yes,
  no,
  maybe
};

typedef enum dependent disambiguation_policy(Node*, Node*);
disambiguation_policy policies[N] = {
   points_to_information,
   address_difference,
   base_address_test,
   symbolic_address_comparison,
   induction_variable_test
};

/* Memory disambiguation engine */

forall (t is tokenEdge(n1, n2)) {
  for (i=0; i < N; i++) {
    switch ((policies[i])(n1, n2)) {
      case yes: /* clearly dependent */
        break;
      case no:  /* clearly independent */
        remove_token_edge(n1, n2);
        break;  /* next pair */
      case maybe:
        continue;
    }
  }
}
```

Figure 4.13: *Pseudo-code for memory disambiguation mechanism implementation.*

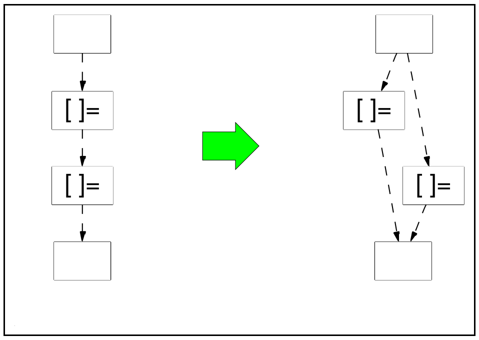


Figure 4.14: *Removing one unnecessary token edge when memory disambiguation can prove that the two stores do not alias.*

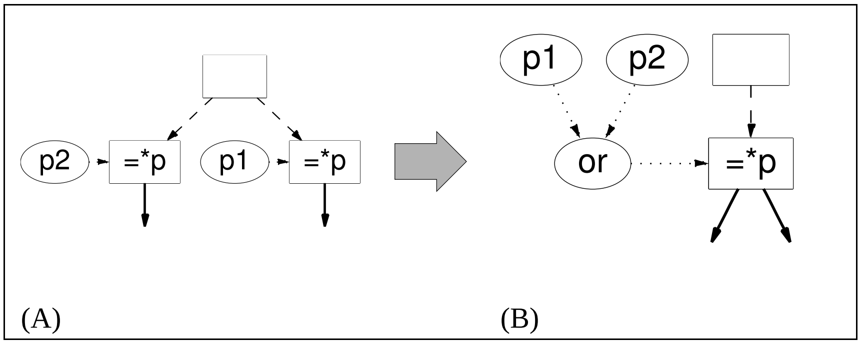


Figure 4.15: *(A) Load operations from the same address and having the* same token source *can be (B) coalesced — the predicate of the resulting load is the disjunction of the original predicates.*

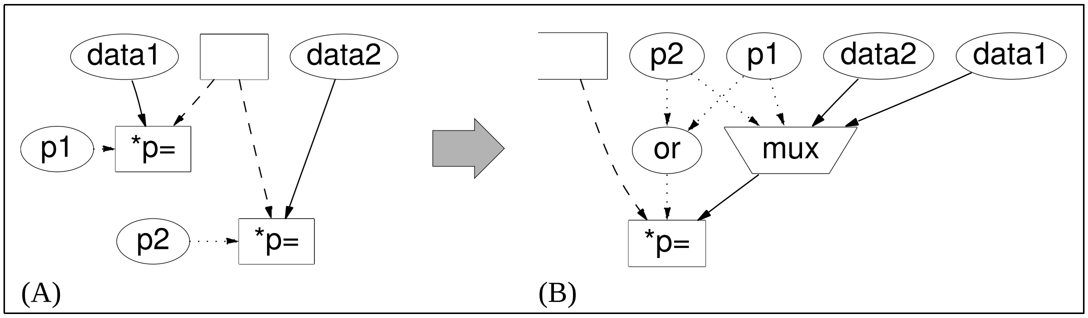


Figure 4.16: *(A) Store operations at the same address and having the* same token source *can be (B) coalesced — the predicate of the resulting store is the disjunction of the original predicates. A multiplexor selects the correct data.*

load merging and redundant store merging (subsuming partial redundancy elimination and global common-subexpression elimination), dead store removal, load-after-store removal (a form of register promotion). Token edges summarize simultaneously both control-flow and dependence information, and make it easy for the compiler to focus its optimizations on interfering memory operations. These transformations are implemented simply as term-rewriting rules, much as depicted in the associated figures.

The optimizations described in this section all assume aligned memory accesses. All are applicable only when the optimized memory operations access the same address and manipulate the same amount of data (i.e., they do not handle the case of a store word followed by a load byte). Testing for address equality is trivial in an SSA representation: two accesses are made to the same address if their address input wire comes from the same source node.

4.4.3.1 Merging Repeated Memory Operations

This transformation generalizes GCSE, PRE and code hoisting for memory accesses, by merging multiple accesses to the same memory address into one single operation. It is a simple term-rewriting optimization; the transformation is applied to loads as shown in Figure 4.15 and to stores as shown in Figure 4.16. If the two predicates originate at the same node, this optimization becomes simply CSE for memory operations. It is easy to see how this transformation can be applied to coalesce more than two operations having the

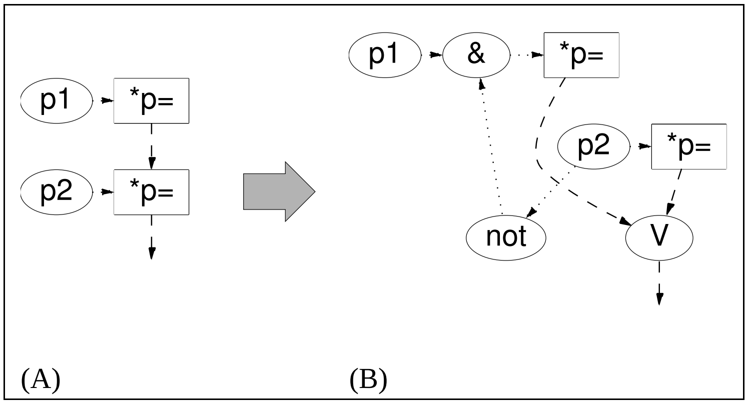


Figure 4.17: *Redundant store-after-store removal: (A) a store immediately preceding another store having the same address should execute only if the second one does not. (B) After this transformation the two stores can never occur at the same time and therefore the token edge connecting them can be safely removed.*

same token source.

As a sanity check, note that when merging two loads into one, the resulting pattern implies that the two loaded values must be identical. This is easy to ascertain from the token graph: since the two loads have the same token source, there is no intervening store between them and the state of the referred location is the same for both (i.e., the state at the end of the execution of their immediate predecessor). Therefore they *must* produce the same result.

An important by-product of this optimization is that the token network is simplified. The operations which used to receive tokens from each of the coalesced operations will now all have an unique token source (the result of the coalescing), and therefore will become candidates for further applications of this transformation.

If we transpose these optimizations in the CFG framework, creating an operation with a predicate $p_1 \vee p_2$ is the counterpart of *lifting* the partially redundant code to a basic block that dominates the basic blocks having predicates p_1 and p_2 and that is post-dominated by them. In general such a block may not even exist in the CFG, and therefore for optimal PRE new control flow might need to be created.

4.4.3.2 Redundant Store Removal

Figure 4.17 depicts the store-before-store code transformation (used in the example in Figure 4.11C). The goal of this transformation is to ensure that at run time the first store is executed only if the second one is not, since the second one will overwrite the result. If the predicate of the prior stores implies the predicate of the following store (which happens if the second post-dominates the first), the first predicate will become constant “false.” If the Boolean manipulation performed by the compiler can prove this fact, the first store can be eliminated using the rule from Figure 4.5. Notice that, if the predicate is false, but the Boolean manipulation cannot simplify it, the store operation nevertheless does not occur at run-time.

This transformation is easily generalized to handle one store S followed by (i.e., connected directly using token edges) multiple stores $S_i, i \in \{1 \ldots n\}$ to the same address. If the predicate of S is p and the predicate of each S_i is p_i, then the optimization replaces p with $p \wedge \neg(\vee_i p_i)$, i.e., the S should be executed only if none of the S_i is. If the set of operations $\{S_i : i \in \{1 \ldots n\}\}$ post-dominates S (as formally defined in [Gup92]), then $\vee_i p_i \Rightarrow p$, and therefore p becomes constant “false”, leading to the elimination of S.

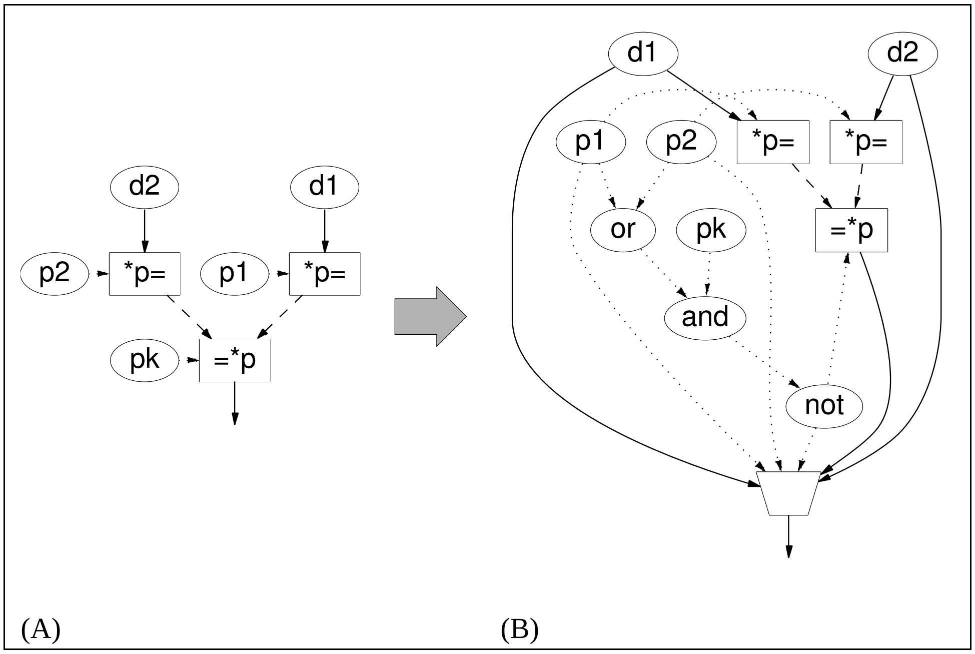


Figure 4.18: *Removal of (A) a load following multiple stores at the same address. (B) The stored value is directly forwarded if any of the stores occurs, otherwise the load has to be performed.*

4.4.3.3 Load After Store Removal (Register Promotion)

In Figure 4.18 we show how loads following a set of store operations at the same address can directly bypass the stored value (Figure 4.11B shows an application of this transformation). This transformation ensures that if either of the stores is executed (i.e., its predicate is dynamically "true"), then the load is not executed. Instead, the stored value is directly forwarded. The load is executed only when none of the stores takes place. This is achieved by using the predicates in a manner similar to the redundant store transformation above. Additionally, a multiplexor is constructed to forward data from the correct store or from the load if none of the stores is executed.

Note that it is *guaranteed* that two of the stores cannot execute simultaneously, since there is no token edge between either two stores (such an edge would make the token graph not be transitively reduced). If the stores collectively dominate the load, the latter is completely removed by acquiring a "false" predicate, and the mux is simplified by the Figure 4.4(b) transformation.

4.4.3.4 The Cycle-Free Condition

Optimizations involving memory operations must properly handle the controlling predicates: an optimization should not create cycles in the program representation. For example, when merging two loads, a cycle may be created if the predicate of one of the loads is a function of the result of the other load, as in Figure 4.19. Such a pattern is produced by code such as: '`if (*p) a+=*p`'.

Testing for the cycle-free condition is easily accomplished with a reachability computation that ignores the back-edges. By caching the results of the reachability computation for a batch of optimizations, and performing the optimizations in reverse topological order (so that the reachability does not change for operations which have not been processed yet), the amortized cost of reachability testing remains linear.

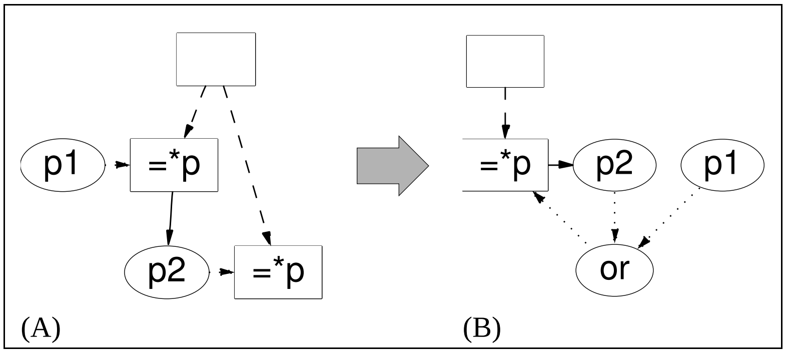


Figure 4.19: *(A) Redundancy elimination operations may not be applied if they lead to cycles in the graph as in (B). The edge *p→p2 in this figure stands for an arbitrary simple path.*

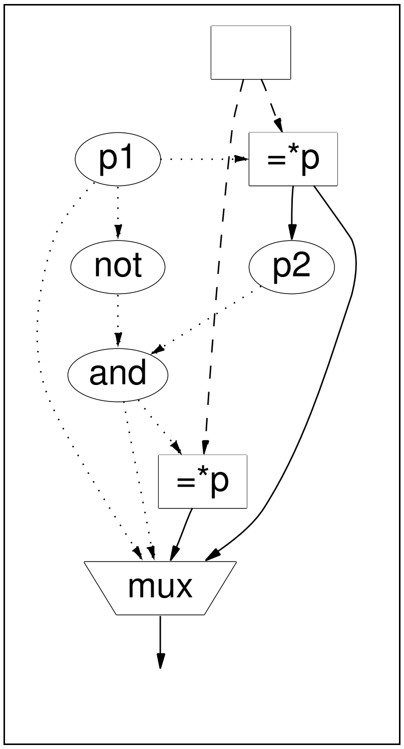


Figure 4.20: *Optimization of the pattern in Figure 4.19(A).*

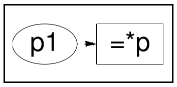


Figure 4.21: *Optimization of the pattern in Figure 4.19(A) and Figure 4.20 when* $p_2 \Rightarrow p_1$.

Patterns which would lead to cycles can still be optimized (and brought to a dynamically optimal form) by eliminating redundancy as shown in Figure 4.20. This optimization is closely related to our new algorithm for register promotion from Section 4.6. The idea is to execute the second load only if the first one does not take place;[8] the predicate controlling the second load becomes $p_2 := p_2 \wedge \neg p_1$. A mux is used to select the output of either the first or the second load. Note that if $p_2 \Rightarrow p_1$ then one of the mux and load controlling predicates becomes constant "false", and the whole structure reduces (after mux and false predicate simplifications) to the one in Figure 4.21. Note that a pattern such as '`if (*p) a+=*p`' is simplified in this way, leading to the execution of a single load at run-time.

4.4.3.5 Comments

Let us make several comments about the optimizations presented in this section:

- They are all very simple to implement. The reachability test in a DAG is trivial (Table 4.1 indicates that it requires 127 lines). All the complexity is in managing the cache of reachability relations properly. Once the reachability test passes the cycle-free condition, the transformation is a simple generalization of term-rewriting (it is not exactly term-rewriting, since it can match patterns with arbitrarily many operations). Only 235 lines of code are required to implement all of them.

- As indicated when presenting each optimization, some optimizations are more drastic when certain dominance relationships between operations occur. (For example, if a store dominates the load to which it forwards the value, the load is completely eliminated.) However, note that in describing these optimizations *we did not need to make any control dependence tests*. The dominance relationship is entirely encoded in the predicates, and partially in the token edges, and therefore there is no need to answer complex queries about the graph structure in order to decide when the optimizations apply. The importance of this encoding lies not in the savings it provides for computing the control relationships, but because the control relationship does not need to be *re-computed* when the program is optimized. For example, as we commented above, "or"-ing to predicate is the same as lifting an operation to a new basic block, which may not exist in the original CFG. In a CFG-based framework on each change of the CFG the control relationship has to be updated, a rather complex endeavor.

- An interesting insight is that, at least for acyclic graphs, the dominance relationships (even generalized dominance and post-dominance) are encoded compactly in the basic-block predicates. Implication is the same as testing (post)dominance, predicate or-ing is the same as creating a basic block control-equivalent to a given set of blocks, the negation of a predicate corresponds to the exclusion of all paths going through a basic block, etc.

- In the same vein, we have transformed CFG manipulations into Boolean manipulations. To really take advantage of the whole power of this method the compiler should feature a rather sophisticated Boolean simplification engine. (For example, if the predicate expression guarding a load is constant "false" the load is never executed dynamically. However, if the compiler is able to prove statically that the expression is "false", it can remove the load completely, and therefore enable further optimizations due to the simplification of the token network.) From a computational complexity point of view it is not clear that the Boolean encoding is a good deal, since Boolean minimization is NP-hard. However, from a pragmatic point of view, it seems to be a worthwhile representation.

[8] Since there is a true dependence between the two loads, they are ordered in a topological sort and we can now speak of the first and the second.

- The structure token network enables powerful inferences about memory state. For example, as we have discussed above, when merging two parallel loads with the same token source we *know* that they will load the same value. Similar arguments can be made about the correctness of the other transformations. Note that the token network enables us to reason about the contents of the accessed location independently on what goes on in the rest of memory.

- This brings about a final point: the use of strong memory disambiguation is a prerequisite for making these algorithms useful. Unfortunately, in general good disambiguation is prohibitive in C. When using hand-annotations to indicate pointer independence the effectiveness of the above transformations increases substantially (Section 7.3).

- Some of these optimizations seem to have a negative effect on the length of the critical path. For example, when merging two parallel loads, the merged load predicate needs both original predicates to be executed. This seems to make the critical path the worse of the two initial paths. Two factors alleviate the impact of this transformation. First, a lenient implementation Section 6.4 of the Boolean operation will only increase the critical path by the delay of the "or": if one input of the "or" is "true", it can immediately generate a "true" output. Second, memory operations are issued to the load-store queue (LSQ; see Section 6.5.3) even without the predicates being known. The LSQ can do dynamic memory disambiguation using run-time addresses and execute speculatively load operations, without knowing their predicate.

4.4.4 Experimental Evaluation

In this section we evaluate the effectiveness of our memory redundancy elimination algorithms. We use programs from the Mediabench, SpecInt95 and SpecInt2K benchmark suites.

4.4.4.1 Static Measurements

Figure 4.22 shows that on average 8.8% of the loads and 8.6% of the static memory operations are removed from the program by these optimizations. Notice we can not directly compare these metrics with compilers such as gcc since CASH can use an unbounded number of registers, while gcc must sometimes spill registers to memory, increasing memory traffic.

4.4.4.2 Dynamic Measurements

We evaluated the effectiveness of our optimizations by evaluating the reduction in memory traffic. Figure 4.23 shows the number of removed memory accesses for each benchmark. The average reduction is 2.6% for loads and 2.4% for stores. There are three ways in which memory operation removal can increase the performance of a program:

- By reducing the number of memory operations executed

- By overlapping many memory operations (increasing traffic bandwidth) through loop pipelining

- By creating more compact schedules and issuing the memory operations faster

Finally, Figure 4.24 shows the impact of these optimizations on the end-to-end performance. While the average performance increase is only about 2.44%, the improvement is likely to increase as memory

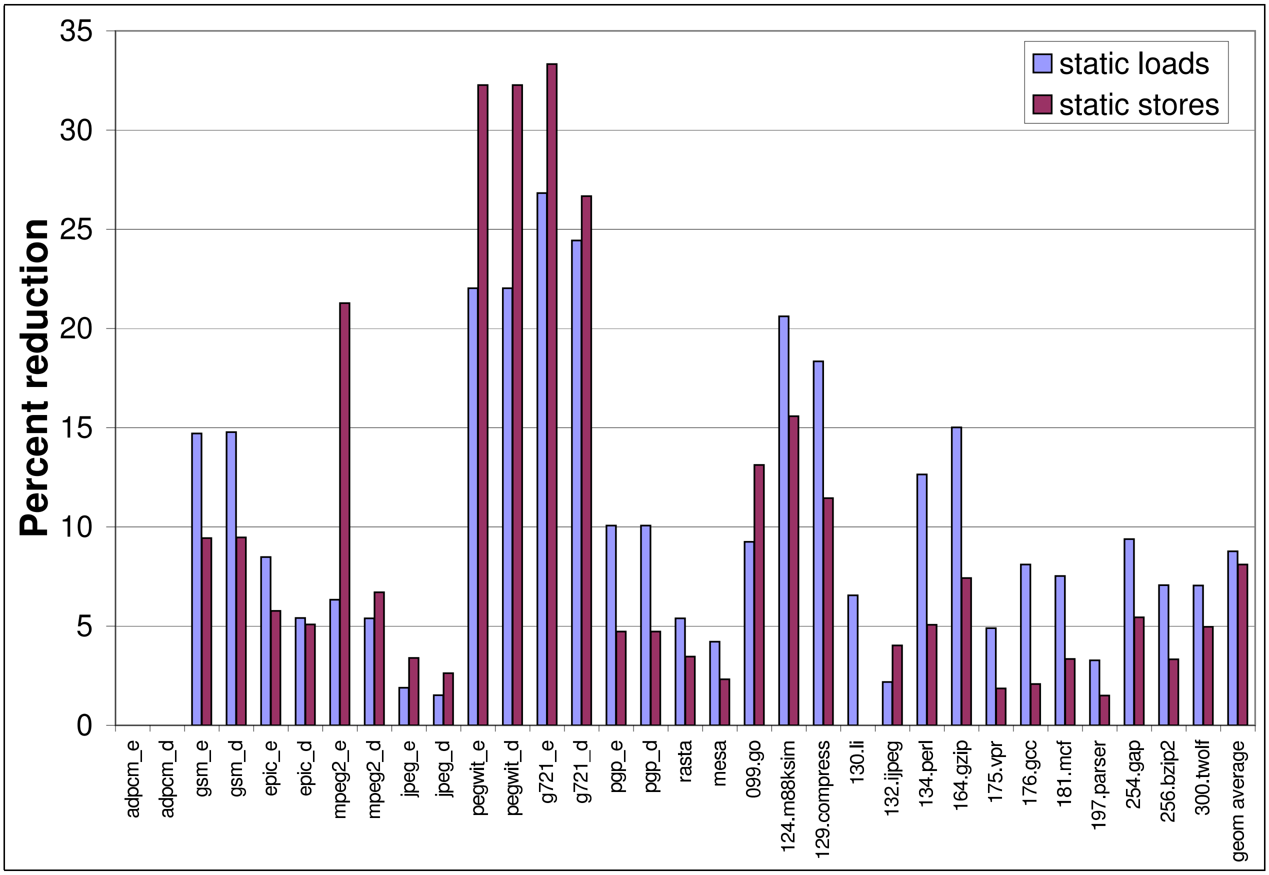


Figure 4.22: *Static reduction of memory accesses, in percentages. The baseline is obtained with CASH and only scalar optimizations (memory optimizations turned off). The optimized program has the scalar replacement algorithm described in Section 4.6 turned off.*

becomes relatively slower and farther. We will also revisit those numbers after presenting a new register promotion, which can remove loads and stores from loops with control flow, the subject of the Section 4.6. Interestingly, the correlation with the number of removed operations is not very strong. The biggest peaks on the two graphs do not correspond to each other.

4.5 Pipelining Memory Accesses in Loops

4.5.1 Motivation

Pipelining loops is especially important to fully realize the performance of dataflow machines. See Section 6.7 for a discussion of dataflow software pipelining. The optimizations presented in this section increase the opportunities for pipelining by removing dependencies between memory operations from different loop iterations and by creating structures that allow them to execute in parallel.

The key to increasing the amount of pipeline parallelism available is to correctly handle memory synchronization across loop iterations. We do this by carrying fine-grained synchronization information across loop iterations using several merge-eta token loops. Figure 4.25 illustrates schematically how pipelining is enabled by fine-grained synchronization. In Figure 4.25(A) is a schematic code segment which reads

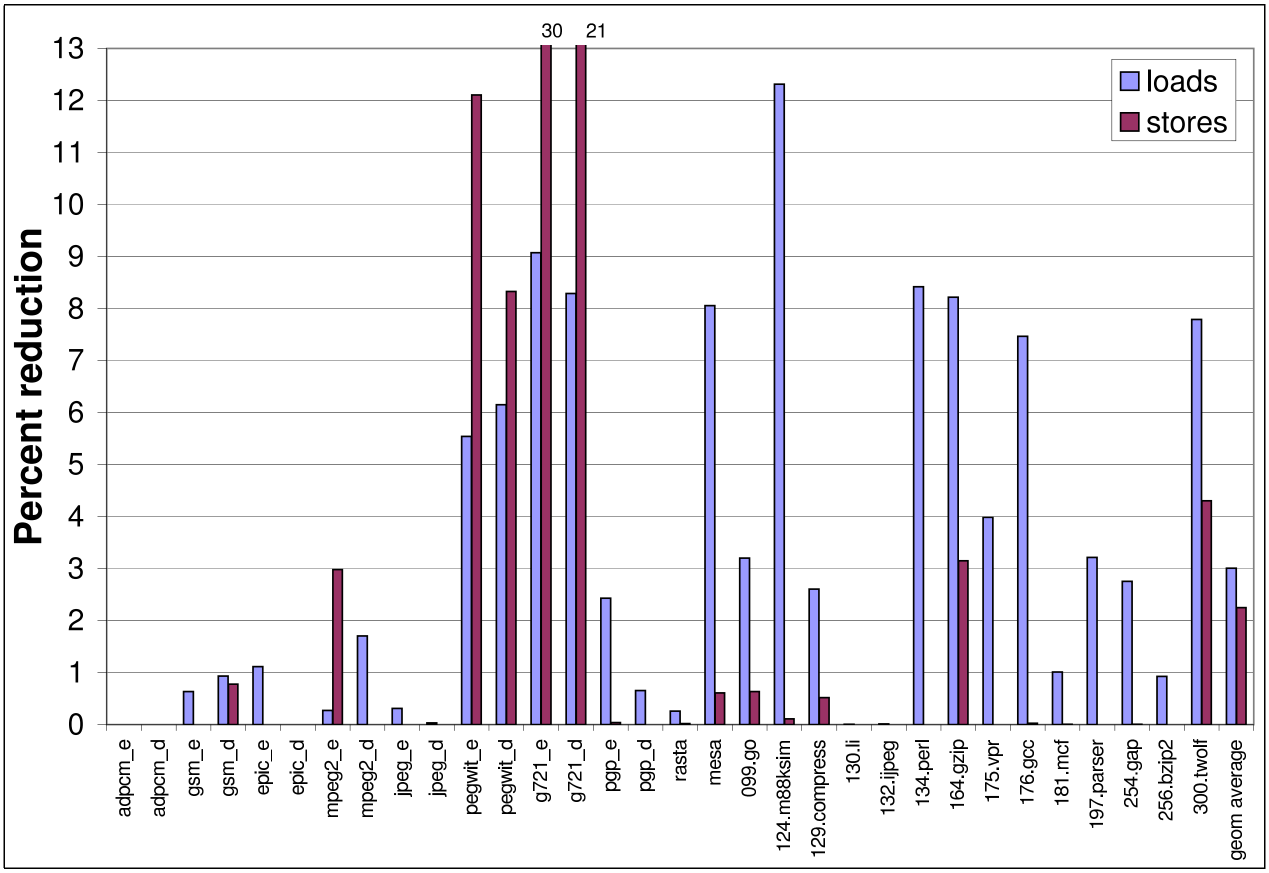


Figure 4.23: *Reduction of dynamic memory traffic in percentages.*

a source array, computes on the data and writes the data into a distinct destination array. Figure 4.25(B, left) a traditional implementation which executes the memory operations in program order: the reads and writes are interleaved. Meanwhile Figure 4.25(B, right) shows how separately synchronizing the accesses to the source and destination array "splits" the loop into two separate loops which can advance at different rates. The "producer" loop can be several iterations ahead of the "consumer" loop, filling the computation pipeline.

The optimizations described in this section all manipulate just the token network, trying to decompose it into several strongly connected components, which can then advance at different paces. These optimizations will be useful however only if the scalar part of the computation does not actually enforce a lock-step execution of the loops. If the scalar network involving the memory accesses is a single strongly connected component, then the benefit of applying the transformations in this section is much reduced.

4.5.2 Fine-Grained Memory Tokens

Figure 4.26 shows an example loop and its token network. Notice that each merge-eta circuit is associated with a particular set of memory locations. There are circuits for `a` and `b` and one, labeled `nonlocals`, which represents all anonymous memory objects. The merge node tagged with `a` represents all accesses made to `a` in the program *before* the current loop iteration, while the eta represents the accesses *following* the current loop iteration.

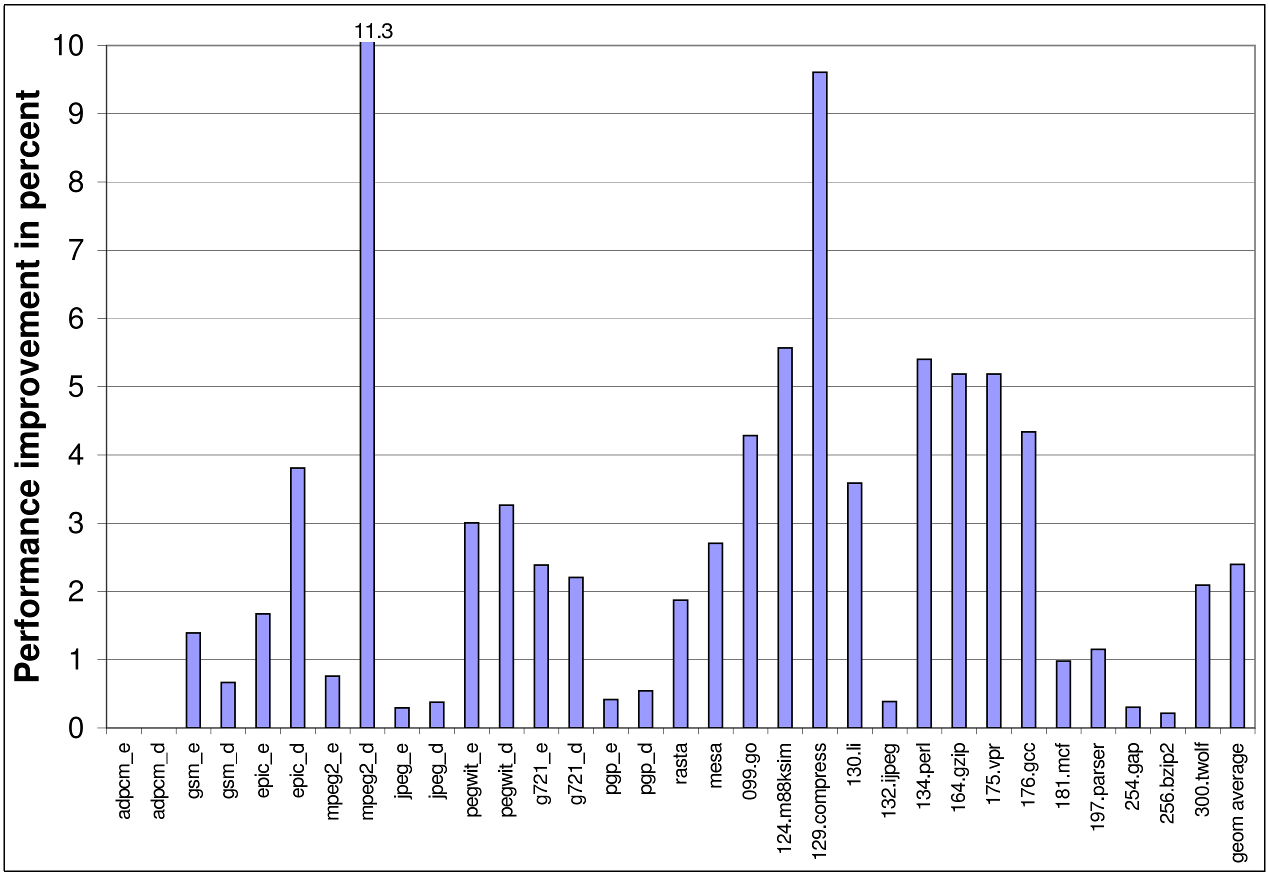


Figure 4.24: *Reduction of execution cycles, in percentages.*

The circuit was generated using the following algorithm:

- The read-write sets of all pointers in the loop body are analyzed. For the example in Figure 4.26, the read-write sets are as follows:

`a`→ {`a[]`}	pointer `a` points to array `a[]`
`b`→ {`b[]`}	
`p`→ {`a[],b[],nonlocals`}	`p` can point to anything

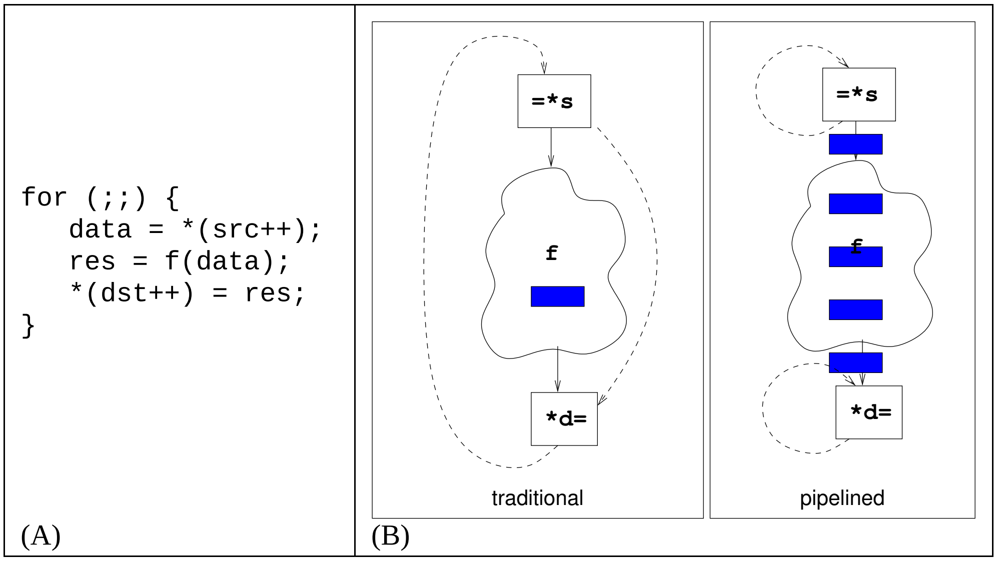
  The objects in the read-write sets are `a[]`, `b[]` and `nonlocals`.

- A merge-eta cycle is created for each object.

- The merge and eta are treated as writes to the corresponding objects.

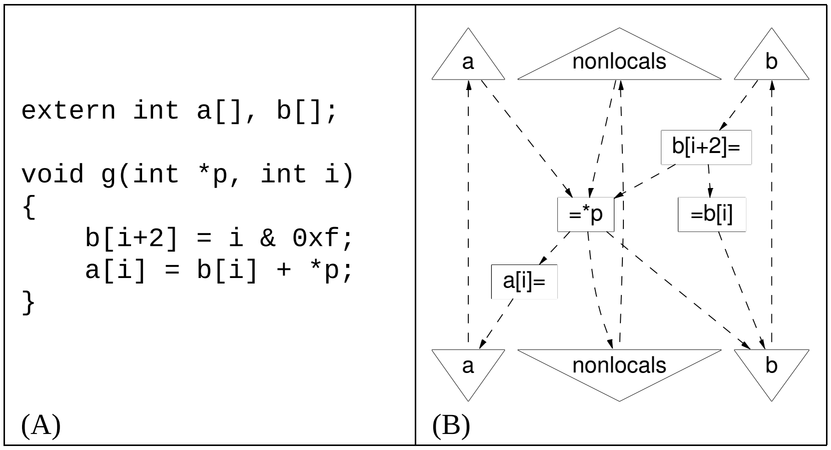
- The loop body is synthesized as straight-line code, as if preceded by all merges and succeeded by all etas.

The token-insertion algorithm for building loops treats the merge and eta operations as writes to the corresponding object(s) when building the token edges. Doing so ensures that all operations accessing the object in the i-th iteration will complete before the $i+1$-th iteration can start.



```
for (;;) {
   data = *(src++);
   res = f(data);
   *(dst++) = res;
}
```

Figure 4.25: *(A) Sample code reading and writing to separate buffers. (B) A traditional implementation performs accesses to the source and destination buffers in lockstep. A pipelined implementation synchronizes separately the accesses to the source and destination buffers, enabling software pipelining.*



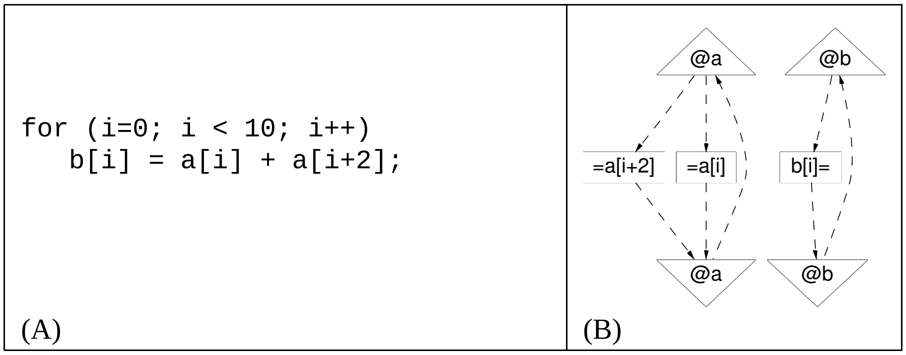
```
extern int a[], b[];

void g(int *p, int i)
{
    b[i+2] = i & 0xf;
    a[i] = b[i] + *p;
}
```

Figure 4.26: *Implementation of memory synchronization in loops.*

4.5.3 Using Address Monotonicity

Treating the token-carrying mu and eta as "writes" may provide excessive synchronization between accesses in different iterations. There is no need to synchronize writes to different addresses (it is even less necessary to do so for the reads of a). For example, all writes to b in Figure 4.27 are to different addresses. CASH attempts to discover such accesses, whose addresses vary strictly monotonically from one iteration to the next, using the induction variable analysis from Section 4.3.4.

If a token loop contains only monotone accesses, it is optimized by splitting it into *two* parallel loops enabling parallel accesses to all locations: (1) a *generator* loop, which generates tokens to enable the



```
for (i=0; i < 10; i++)
   b[i] = a[i] + a[i+2];
```

Figure 4.27: *Sample program and naive implementation of a loop containing monotone accesses to arrays* a *and* b*.*

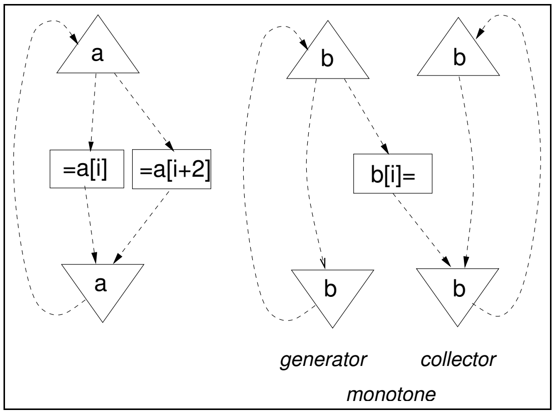


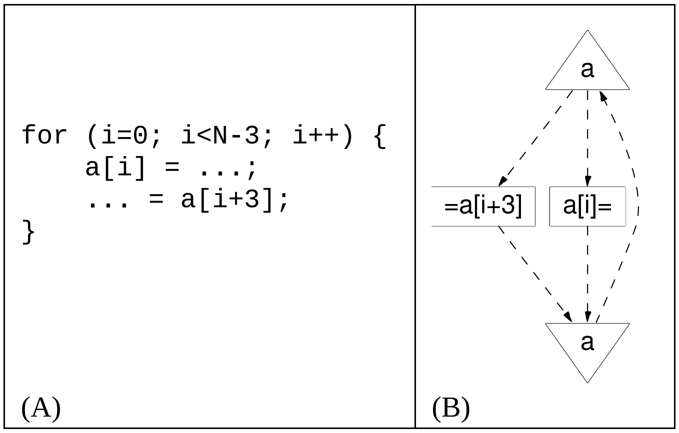
Figure 4.28: *The loop in Figure 4.27 after optimizing for address monotonicity.*

operations for all the loop iterations, allowing operations from multiple iterations to issue simultaneously, and (2) a *collector* loop, which collects the tokens from operations in all iterations. This ensures that the loop terminates only when all operations in all iterations have occurred. The result after the optimization of Figure 4.27 is shown in Figure 4.28.

4.5.4 Dynamic Slip Control: Loop Decoupling

In this section we present a novel form of loop parallelization that is particularly effective for Spatial Computation. This transformation, *loop decoupling*, is remotely related to loop skewing and loop splitting. Analogously to the optimization for monotone accesses, it "vertically" slices a loop into multiple independent loops which are allowed to *slip* with respect to each other (i.e., one can get many iterations ahead of the other). The maximum slip is derived from the *dependence distance* and ensures that the loops slipping ahead do not violate any program dependence. To dynamically control the amount of slip a new operator is used, a *token generator*.

We illustrate decoupling on the circuit in Figure 4.29, obtained after the previously described optimizations are applied to the shown program. Dependence analysis (based on induction variable analysis, as described in Section 4.3.4) indicates that the two memory accesses are at a fixed distance of three iterations. Loop decoupling separates the two accesses into two independent loops by creating a token for each of



```
for (i=0; i<N-3; i++) {
    a[i] = ...;
    ... = a[i+3];
}
```

Figure 4.29: *Program and its implementation before loop decoupling.*

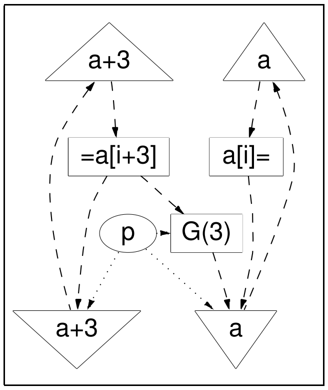


Figure 4.30: *The program from Figure 4.29 after loop decoupling. We have also depicted p, the loop-controlling predicate.*

them. The `a[i+3]` loop can execute as quickly as possible. However, the loop for `a[i]` can be ahead only three or fewer iterations of the `a[i+3]` loop because otherwise it would violate the dependences. To dynamically bound the slip between these two loops a *token generator* operation which can generate three tokens is inserted between the two loops. The resulting structure is depicted in Figure 4.30. The token generator can "instantly" generate three tokens. Subsequently it generates additional tokens upon receipt of tokens at the input.

More precisely, a token generator G(n) has two inputs: a predicate and a token, and one output: a token (see also Section 3.2.3.5). n is a positive integer, the dependence distance between the two accesses. The token generator maintains a counter, initialized with n. The predicate input is the loop-controlling predicate. On receipt of a "true" predicate, the generator decrements the counter, and if the counter is positive, it emits a token. On receipt of a token it increments the counter. On receipt of a "false" predicate (indicating completion of the loop), the predicate input is not acknowledged until the token counter becomes n again (indicating that all input tokens have been received). There is a strong similarity between the token generator operation and Dijkstra semaphores [Dij65]. In the general case, for n different addresses accessed, loop decoupling creates $n - 1$ token generators, chained as shown schematically in Figure 4.31.

The `a[i]` loop must receive tokens from the `a[i+3]` loop in order to make progress. It may receive

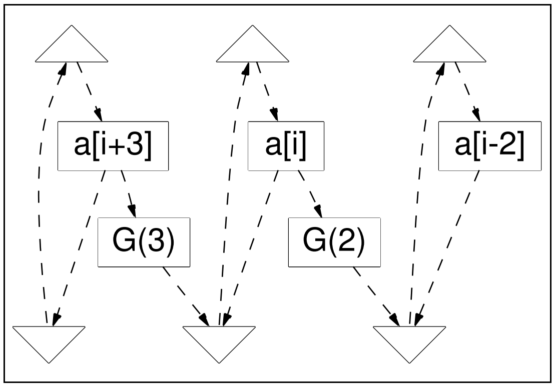


Figure 4.31: *An instance of decoupling involving three components.*

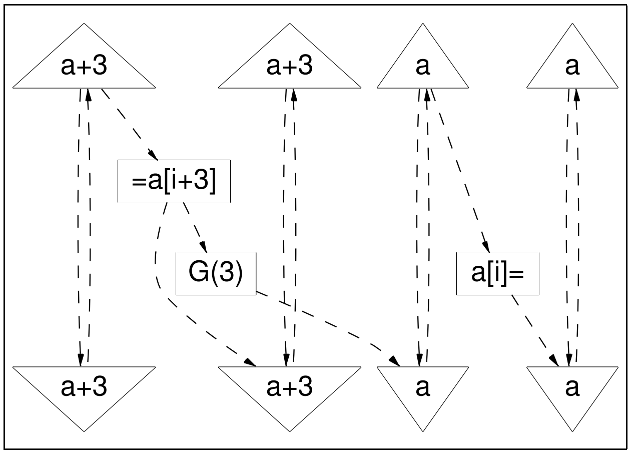


Figure 4.32: *The program in Figure 4.29 after loop decoupling and monotone memory address optimization.*

the first three tokens before any iteration of the latter has completed, directly from the token generator, so it can slip up to three iterations ahead. Afterwards, it will receive a new token enabling it to make progress only after the `a[i+3]` loop completes one iteration. In contrast, the `a[i+3]` loop can slip an arbitrary number of iterations ahead (the token generator stores the extra tokens generated by this loop in its internal counter).

Notice that after loop decoupling each of the resulting loops contains only monotone addresses, so it can be further optimized as described in Section 4.5.3. The final result is shown in Figure 4.32.

The idea of explicitly representing the dependence distances in a program can be found in Johnson's paper "An executable representation of distance" [JLP91]. While that paper gives a general scheme for representing dependence distances in nested loops as functions of multiple loop indices, it does not discuss how the resulting representation can be used to optimize the program. Our scheme can also be generalized to deal with nested loops, although our current implementation handles only innermost loops. The incarnation of the dependence distance in the "token generator" operator, used to dynamically control the slip between decoupled loop iterations is a new idea, which, we believe, is indeed an *executable representation of distance*.

4.6 Optimal Inter-Iteration Register Promotion

In this section we revisit the classical problem of scalar replacement of array elements and pointer accesses. We generalize the state-of-the-art algorithm, by Carr and Kennedy [CK94], to optimally handle a combination of both conditional control-flow and inter-iteration data reuse. (In this text we are not considering at all speculative promotion, which has been extensively studied since then. However, the technique we describe is also applicable to speculative scalar promotion as well.) The basis of our algorithm is to make the dataflow availability information precise by computing and using it at runtime.

Our algorithm operates within the same assumptions of the classical one, that is, perfect dependence information, and has the same limitations, i.e., increased register pressure. It is, however, optimal in the following sense: (assuming the availability of enough scalar registers) within each code region where scalar promotion is applied, each memory location is read at most once and written at most once.

The basis of our algorithm is to make the dataflow availability information precise using a technique we call Statically Instantiate and Dynamically Evaluate (SIDE). In SIDE the compiler inserts explicit code to evaluate the dataflow information at runtime. As the computational bandwidth of processors increases, these kinds of optimizations may become more advantageous. In the case of register promotion, the benefit from removing expensive memory operations outweighs the increase in scalar computations to maintain the dataflow information. Applying this technique to other dataflow analyses may be a fruitful avenue of research.

4.6.1 Preliminaries

The goal of scalar replacement (also called register promotion) is to identify repeated accesses made to the same memory address, either within the same iteration or across iterations, and to remove the redundant accesses (CASH performs promotion within the innermost loop bodies. But the ideas we present are applicable to wider code regions as well). The state-of-the-art algorithm for scalar replacement was proposed in 1994 by Steve Carr and Ken Kennedy [CK94]. This algorithm handles very well two special instances of the scalar replacement problem: (1) repeated accesses made within the same loop iteration in code having arbitrary conditional control-flow; and (2) code with repeated accesses made across iterations *in the absence of conditional control-flow*. For (1) the algorithm relies on PRE, while for (2) it relies on dependence analysis and rotating scalar values. However, that algorithm cannot handle a combination of both conditional control-flow and inter-iteration reuse of data.

Here we present a very simple algorithm which generalizes and simplifies the Carr-Kennedy algorithm in an optimal way. The optimality criterion that we use throughout this section is the number of dynamically executed memory accesses. After application of our algorithm on a code region, no memory location is read more than once and written more than once in that region. Also, after promotion, no memory location is read or written if it was not so in the original program (i.e., our algorithm does not perform speculative promotion). Our algorithm operates under the same assumptions as the Carr-Kennedy algorithm. That is, it requires perfect dependence information to be applicable. It is therefore mostly suitable for FORTRAN benchmarks. Since CASH is a C compiler, we have tested its applicability only to C programs and we have found numerous instances where it is applicable as well.

For the impatient reader, the key idea is the following: for each value to be scalarized, the compiler creates a 1-bit runtime flag variable indicating whether the scalar value is "valid." The compiler also creates code which dynamically updates the flag. The flag is then used to detect and avoid redundant loads and to indicate whether a store has to occur to update a modified value at loop completion. This algorithm ensures that only the first load of a memory location is executed and only the last store takes place. This algorithm is a particular instance of a new general class of algorithms: it transforms values customarily used only at compile-time for dataflow analysis into dynamic objects. Our algorithm instantiates *availability* dataflow information into run-time objects, therefore achieving dynamic optimality even in the presence of constructs which cannot be statically optimized.

We introduce the algorithm by a series of examples which show how it is applied to increasingly complicated code structures. We start in Section 4.6.3 by showing how the algorithm handles a special case, that of memory operations from loop-invariant addresses. In Section 4.6.4.1 we show how the algorithm optimizes loads whose addresses are induction variables. Finally, we show how stores can be treated optimally in Section 4.6.4.4. In Section 4.6.5.2 we show how the algorithm is implemented in Pegasus. Although a CFG variant is simpler to implement, Pegasus simplifies the dependence analysis required to determine whether promotion is applicable. Special handling of loop-invariant guarding predicates is discussed in Section 4.6.6. Finally, in Section 4.6.8, we quantify the impact of an implementation of this algorithm when applied to the innermost loops of a series of C programs.

This section makes the following new research contributions:

- It introduces a linear-time[9] term-rewriting algorithm for performing inter-iteration register promotion in the presence of control-flow.
- It shows that this new optimization is a program transformation which materializes compiler dataflow values as dynamic, run-time variables. This suggests a new research direction for adapting other dataflow algorithms, creating dynamically-optimal code for constructs which cannot be statically made optimal.
- It describes register promotion as implemented in Pegasus, showing how it takes advantage of the token network for effective dependence analysis.

4.6.2 Conventions

We present all the optimizations examples as source-to-source transformations of schematic C program fragments instead of showing how they manipulate Pegasus. For simplicity of the exposition we assume that we are optimizing the body of an innermost loop. We also assume that none of the scalar variables in our examples have their address taken. We write `f(i)` to denote an arbitrary expression involving `i` which has no side effects (but *not* a function call). We write `for(i)` to denote a loop having `i` as a basic induction variable.

For pedagogical purposes, the examples we present all assume that the code has been brought into a canonical form through the use of *if-conversion* [AKPW83], such that each memory statement is guarded by a predicate; i.e., the code has the shape in Figure 4.33. Our algorithms are easily generalized to handle nested natural loops and arbitrary forward control-flow within the loop body.

4.6.3 Scalar Replacement of Loop-Invariant Memory Operations

In this section we describe a new register promotion algorithm which can eliminate memory references made to loop-invariant addresses in the presence of control flow. This algorithm is further expanded in Section 4.6.4.1 and Section 4.6.4.4 to promote memory accesses into scalars when the memory references have a constant stride.

[9] This time does not include the time to compute the dependences, only the actual register promotion transformation.

```
while (1) {
    if (cond1) statement1;
    if (cond2) statement2;
    ...
    if (cond_break1) break;
    ...
}
```

Figure 4.33: *For ease of presentation we assume that prior to register promotion, all loop bodies are predicated.*

```
for (i)
    *p += i;

-------------

tmp = *p;
for (i)
    tmp += i;
*p = tmp;
```

Figure 4.34: *A simple program before and after register promotion of loop-invariant memory operations.*

```
for (i)
    if (i & 1)
        *p += i;
```

Figure 4.35: *A small program that is not amenable to classical register promotion.*

4.6.3.1 The Classical Algorithm

Figure 4.34 shows a simple example and how it is transformed by the classical scalar promotion algorithm. Assuming `p` cannot point to `i`, the key fact is `*p` always loads from and stores to the same address, therefore `*p` can be transformed into a scalar value. The load is lifted to the loop pre-header, while the store is moved after the loop. (The latter is slightly more difficult to accomplish if the loop has multiple exits going to multiple destinations. Our implementation handles these as well, as described in Section 4.6.5.2.)

4.6.3.2 Loop-Invariant Addresses and Control-Flow

However, the simple algorithm is no longer applicable to the slightly different Figure 4.35. Lifting the load or store out of the loop may be unsafe with respect to exceptions: one cannot lift a memory operation out

```
/* prelude */
tmp = uninitialized;
tmp_valid = false;

for (i) {
    /* load from *p becomes: */
    if ((i & 1) && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }

    /* store to *p becomes */
    if (i & 1) {
        tmp += i;
        tmp_valid = true;
    }
}

/* postlude */
if (tmp_valid)
    *p = tmp;
```

Figure 4.36: *Optimization of the program in Figure 4.35.*

of a loop it if may never be executed within the loop.

However, to optimize Figure 4.35, it is enough to maintain a `valid` bit in addition to the the `tmp` scalar. The `valid` bit indicates whether `tmp` indeed holds the value of `*p`; see Figure 4.36. The `valid` bit is initialized to *false*. A load from `*p` is performed only if the `valid` bit is *false*. Either loading from or storing to `*p` sets the `valid` bit to *true*. This program will forward the value of `*p` through the scalar `tmp` between iterations arbitrarily far apart.

The insight is that it may be profitable to compute dataflow information at runtime. For example, the `valid` flag within an iteration is nothing more than the dynamic equivalent of the *availability* dataflow information for the loaded value, which is the basis of classical PRE. When PRE can be applied statically, it is certainly better to do so. The problem with Figure 4.35 is that the compiler cannot statically summarize when condition `(i&1)` is true, and therefore has to act conservatively, assuming that the loaded value is never available. Computing the availability information at run-time eliminates this conservative approximation. Maintaining and using runtime dataflow information makes sense when we can eliminate costly operations (e.g., memory accesses) by using inexpensive operations (e.g., Boolean register operations).

This algorithm generates a program which is optimal with respect to the number of loads within each region of code to which promotion is applied (if the original program loads from an address, then the optimized program will load from that address exactly once), but may execute one extra store:[10] if the original program loads the value but never stores to it, the `valid` bit will be true, enabling the postlude store. In order to treat this case as well, a `dirty` flag, set on writes, has to be maintained, as shown in

[10] However, this particular program is optimal for stores as well.

```
/* prelude */
tmp = uninitialized;
tmp_valid = false;
tmp_dirty = false;

for (i) {
    /* load from *p becomes: */
    if ((i & 1) && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }

    /* store to *p becomes */
    if (i & 1) {
        tmp += i;
        tmp_valid = true;
        tmp_dirty = true;
    }
}

/* postlude */
if (tmp_dirty)
    *p = tmp;
```

Figure 4.37: *Optimization of store handling from Figure 4.36.*

Figure 4.37.[11]

Note: in order to simplify the presentation, the examples in the rest of the section will not include the `dirty` bit. However, its presence is required for achieving an optimal number of stores.

4.6.4 Inter-Iteration Scalar Promotion

Here we extend the algorithm for promoting loop-invariant operations to perform scalar promotion of pointer and array variables with constant stride. We assume that the code has been subjected to standard induction dependence analysis prior to scalar promotion.

4.6.4.1 The Carr-Kennedy Algorithm

Figure 4.38 illustrates the classical Carr-Kennedy inter-iteration register promotion algorithm from [CCK90], which is only applicable in the absence of control-flow. In general, reusing a value after k iterations requires the creation of k distinct scalar values, to hold the simultaneously live values of `a[i]` loaded for k consecutive values of `i`. This quickly creates register pressure and therefore heuristics are usually used to decide whether promotion is beneficial. Since register pressure has been very well addressed in the literature [CCK90, Muc97, CMS96, CW95], we will not concern ourselves with it anymore in this text.

[11] The `dirty` bit may also be required for correctness, if the value is read-only and the writes within the loop are always dynamically predicated "false."

```
for (i = 2; i < N; i++)
    a[i] = a[i] + a[i-2];

--------------------------------------------

/* pre-header */
a0 = a[0];      /* invariant: a0 = a[i-2] */
a1 = a[1];      /*            a1 = a[i-1] */
for (i = 2; i < N; i++) {
    a2 = a[i];  /*            a2 = a[ i ] */
    a2 = a0 + a2;
    a[i] = a2;

    /* Rotate scalar values */
    a0 = a1;
    a1 = a2;
}
```

Figure 4.38: *Program with no control-flow before and afer register promotion performed by the Carr-Kennedy algorithm.*

A later extension to the Carr-Kennedy algorithm [CK94] allows it to also handle control flow. The algorithm optimally handles reuse of values *within* the same iteration, by using PRE on the loop body. However, this algorithm can no longer promote values *across* iterations in the presence of control-flow. The compiler has difficulty in reasoning about the intervening updates between accesses made in different iterations in the presence of control-flow.

4.6.4.2 Partial Redundancy Elimination

Before presenting our solution let us note that even the classical PRE algorithm (without the support of special register promotion) is quite successful in optimizing loads made in *consecutive* iterations. Figure 4.39 shows a sample loop and its optimization by `gcc`, which does *not* have a register promotion algorithm at all. By using PRE alone `gcc` manages to reuse the load from `ptr2` one iteration later.

The PRE algorithm is unable to achieve the same effect if data is reused in any iteration other than the immediately following iteration or if there are intervening stores. In such cases an algorithm like Carr-Kennedy is necessary to remove the redundant accesses. Let us notice that the use of `valid` flags achieves the same degree of optimality as PRE *within* an iteration, but at the expense of maintaining run-time information.

4.6.4.3 Removing All Redundant Loads

However, the classical algorithm is unable to promote all memory references guarded by a conditional, as in Figure 4.40. It is, in general, impossible for a compiler to check when `f(i)` is true in both iteration `i` and in iteration `i-2`, and therefore it cannot deduce whether the load from `a[i]` can be reused as `a[i-2]` two iterations later.

```
do {
    *ptr1++ = *ptr2++;
} while(--cnt && *ptr2);

------------------------

tmp = *ptr2;
do {
    *ptr1++ = tmp;
    ptr2++;
    if (--cnt) break;
    tmp = *ptr2;
    if (! tmp) break;
} while(1);
```

Figure 4.39: *Sample loop and its optimization using PRE. (The output is the equivalent of the assembly code generated by `gcc`.) PRE can achieve some degree of register promotion for loads.*

```
for (i = 2; i < N; i++)
    if (f{i}) a[i] = a[i] + a[i-2];
```

Figure 4.40: *Sample program which cannot be handled optimally by either PRE or the classical Carr-Kennedy algorithm.*

Register promotion has the goal of only executing the *first* load and the *last* store of a variable. The algorithm in Section 4.6.3 for handling loop-invariant data is immediately applicable for promoting loads across iterations, since it performs a load as soon as possible. By maintaining availability information at runtime, using `valid` flags, our algorithm can transform the code to perform a minimal number of loads as in Figure 4.41. Applying constant propagation and dead-code elimination will simplify this code by removing the unnecessary references to `a2_valid`.

4.6.4.4 Removing All Redundant Stores

Handling stores seems to be more difficult, since one should forgo a store if the value will be overwritten in a subsequent iteration. However, in the presence of control-flow it is not obvious how to deduce whether the overwriting stores in future iterations will take place. Here we extend the register promotion algorithm to ensure that only one store is executed to each memory location, by showing how to optimize the example in Figure 4.42.

We want to avoid storing to `a[i+2]`, since that store will be overwritten two iterations later by the store to `a[i]`. However, this is not true for the last two iterations of the loop. Since, in general, the compiler cannot generate code to test loop-termination several iterations ahead, it looks as if both stores must be performed in each iteration. However, we can do better than that by performing, within the loop, only the store to `a[i]`, which certainly will not be overwritten. The loop in Figure 4.43 does exactly that. The

```
a0_valid = false;
a1_valid = false;
a2_valid = false;
for (i) {
    fi = f(i);

    /* load a[i-2] */
    if (fi && !a0_valid) {
        a0 = a[i-2];
        a0_valid = true;
    }

    /* load a[i] */
    if (fi && !a2_valid) {
        a2 = a[i];
        a2_valid = true;
    }

    /* store a[i] */
    if (fi) {
        a2 = a0 + a2;
        a[i] = a2;
        a2_valid = true;
    }

    /* Rotate scalars and valid flags */
    a0 = a1;
    a1 = a2;
    a0_valid = a1_valid;
    a1_valid = a2_valid;
    a2_valid = false;
}
```

Figure 4.41: *Optimal version of the code in Figure 4.40.*

loop body never overwrites a stored value but may fail to correctly update the last two elements of array `a`. Fortuitously, after the loop completes, the scalars `a0`, `a1` hold exactly these two values. So we can insert a loop postlude to fix the potentially missing writes. (Of course, `dirty` bits should be used to prevent useless updates.)

4.6.5 Implementation

This algorithm is probably much easier to illustrate than to describe precisely. Since the important message was hopefully conveyed by the examples, we will just briefly sketch the implementation in a CFG-based framework and describe in somewhat more detail the Pegasus implementation.

```
for (i) {
    a[i]++;
    if (f(i)) a[i+2] = a[i];
}

---------------------------------------

a0_valid = true;
a0 = a[0];          /* a[i] */
a1_valid = false;
a2_valid = false;
for (i) {
    fi = f(i);

    /* load a[i] */
    if (!a0_valid)
        a0 = a[i];

    /* store a[i] */
    a[i] = a0+1;

    /* store a[i+2] */
    if (fi) {
        a2 = a0;
        a[i+2] = a2;
        a2_valid = true;
    }

    /* Rotate scalars and valid flags */
    a0 = a1;
    a1 = a2;
    a0_valid = a1_valid;
    a1_valid = a2_valid;
    a2_valid = false;
}
```

Figure 4.42: *Program with control-flow and redundant stores and its optimization for an optimal number of loads. The store to* `a[i+2]` *may be overwritten two iterations later by the store to* `a[i]`.

4.6.5.1 CFG-Based Implementation

In general, for each constant reference to a[i+j] (for a compile-time constant j) we maintain a scalar t$_j$ and a valid bit t$_j$valid. Then scalar replacement just makes the following changes:

- Replaces every load from a[i+j] with a pair of statements:
  t$_j$ = t$_j$valid ? t$_j$: a[i+j]; t$_j$valid = true
- Replace every store a[i+j] = e with a pair of statements:

```
a0_valid = true;  /* a[i] */
a0 = a[0];
a1_valid = false;
a2_valid = false;
for (i) {
    fi = f(i);

    /* load a[i] */
    if (!a0_valid)
        a0 = a[i];

    /* store a[i] */
    a[i] = a0+1;

    /* store a[i+2] */
    if (fi) {
        a2 = a0;
        /* No update of a[i+2]: may be overwritten */
        a2_valid = true;
    }

    /* Rotate scalars and valid flags */
    a0 = a1;
    a1 = a2;
    a0_valid = a1_valid;
    a1_valid = a2_valid;
    a2_valid = false;
}

/* Postlude */

if (a0_valid)
    a[i] = a0;
if (a1_valid)
    a[i+1] = a1;
```

Figure 4.43: *Optimal version of the example in Figure 4.42.*

t_j = e; $t_j$valid = true.

Furthermore, all stores except the generating store[12] are removed. Instead compensation code is added "after" the loop: for each t_j append a statement if ($t_j$valid) a[i+j] = t_j.

Complexity: the algorithm, aside from the dependence analysis, is linear in the size of the loop.

Correctness and optimality: are easy to argue. They both follow from the following loop invariant:

[12] According to the terminology in [CCK90], a generating store is the one writing to a[i+j] for the smallest j promoted.

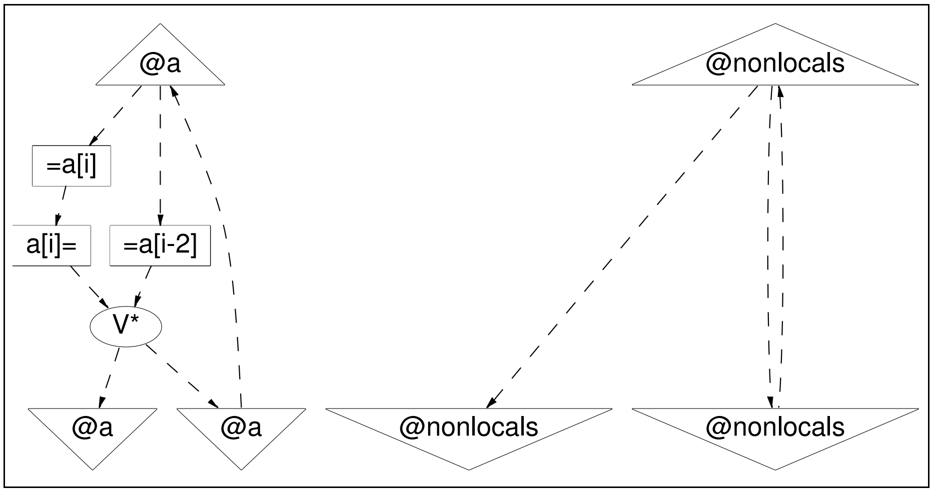


Figure 4.44: *Token network for the Pegasus representation of the loop in Figure 4.40.*

Invariant: the `t`$_j$`valid` flag is true if and only if `t`$_j$ represents the contents of the memory location it scalarizes.

4.6.5.2 Register Promotion in Pegasus

We sketch the most important analysis and transformation steps carried out by CASH for register promotion. Although the actual promotion in Pegasus is slightly more complicated than in a CFG-based representation (because of the need to maintain ϕ-nodes), the dependence tests used to decide whether promotion can be applied are much simpler: the graph will have a very restricted structure if promotion can be applied.[13] The key element of the representation is the token edge network whose structure can be quickly analyzed to determine important properties of the memory operations.

We illustrate register promotion on the example in Figure 4.40.

1. The token network for the Pegasus representation is shown in Figure 4.44. Memory accesses that may interfere with each other will all belong to a same connected component of the token network. Operations that belong to distinct components of the token network commute and can therefore be analyzed separately. In this example there are two connected component, corresponding to accesses made to the array `a` and to `nonlocals`. However, since there are no accesses in the `nonlocals` network, it can be ignored.

2. The addresses of the three memory operations in this component are analyzed as described in Section 4.3.4: they are all determined to be induction variables having the same step, 1. This implies that the dependence distances between these accesses are constant (i.e., iteration-independent), making these accesses candidates for register promotion.

   The induction step of the addresses indicates the type of promotion: a 0 step indicates loop-invariant accesses, while a non-zero step, as in this example, indicates strided accesses.

[13] The network encodes relatively simple dependence information. However, as pointed in [CMS96], elementary dependence tests are sufficient for most cases of register promotion.

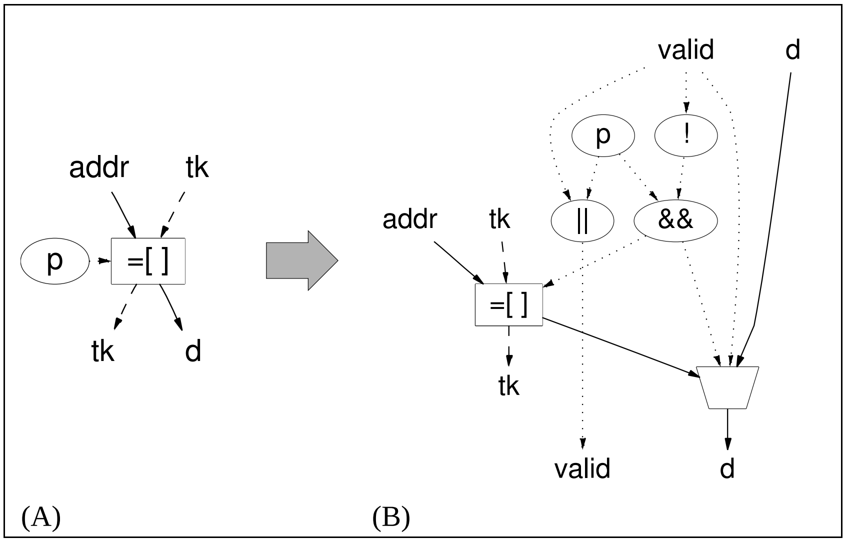


Figure 4.45: *Term-rewriting of loads for register promotion. "d" and "valid" are the register-promoted data and its valid bit respectively.*

3. The token network is further analyzed. Notice that prior to register promotion, memory disambiguation has already proved (based on symbolic computation on address expressions) that the accesses to `a[i]` and `a[i+2]` commute, and therefore there is no token edge between them. The token network for `a` consists of two *strands*: one for the accesses to `a[i]`, and one for `a[i+2]`; the strands are generated at the mu, on top, and joined before the etas, at the bottom, using a combine (V). If and only if all memory accesses within the same strand are made to the same address can promotion be carried.

   CASH generates the initialization for the scalar temporaries and the "valid" bits in the loop preheader, as described in Section 4.6. We do not illustrate this step.

4. Each strand is scanned from top to bottom (from the mu to the eta), term-rewriting each memory operation:

   - Figure 4.45 shows how a load operation is transformed by register promotion. The resulting construction can be interpreted as follows: "If the data is already valid do not do the load (i.e., the load predicate is 'and'-ed with the negation of the valid bit) and use the data. Otherwise do the load if its predicate indicates it needs to be executed." The multiplexor will select either the load output or the initial data, depending on the predicates. If neither predicate is true, the output of the mux is not defined, and the resulting `valid` bit is false.
   - Figure 4.46 shows the term-rewriting process for a store. After this transformation, all stores except the generating store are removed from the graph. The resulting construction is interpreted as follows: "If the store occurs, the data-to-be-stored replaces the register-promoted data, and it becomes valid. Otherwise, the register-promoted data remains unchanged."

5. Code is synthesized to shift the scalar values and predicates around between strands (the assignments $\mathtt{t}_{j-1}$ `=` $\mathtt{t}_j$), as illustrated in Figure 4.47.

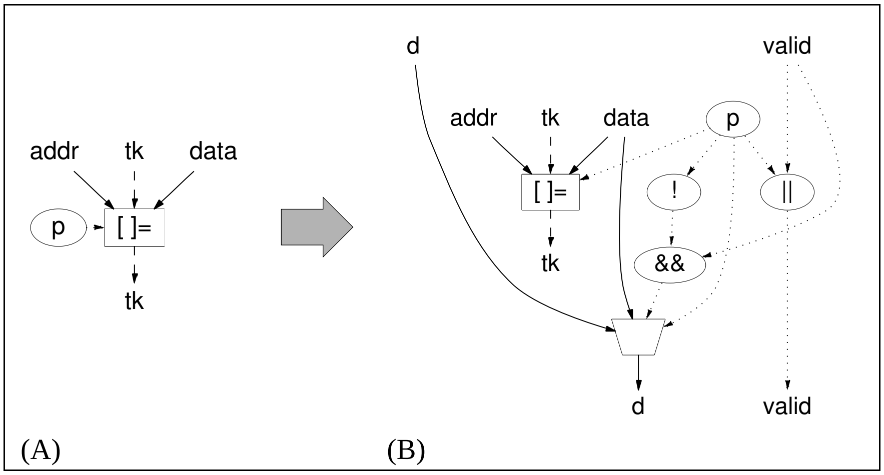


Figure 4.46: *Term-rewriting of stores for register promotion.*

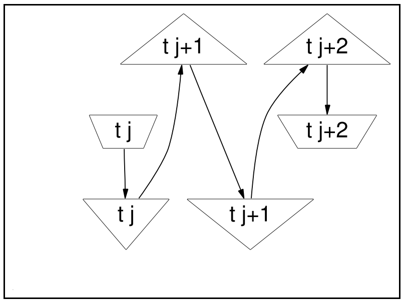


Figure 4.47: *Shifting scalar values between strands. A similar network is used to shift the "valid" bits.*

6. The insertion of a loop postlude is somewhat more difficult in general than a loop prelude, since by definition natural loops have a unique entry point, but may have multiple exits. In our implementation each loop body is completely predicated and therefore all instructions get executed, albeit some are nullified by the predicates. The compensating stores are added to the loop body and executed only during the last iteration. This is achieved by making the predicate controlling these stores to be the loop-termination predicate. This step is not illustrated.

4.6.6 Handling Loop-Invariant Predicates

The register promotion algorithm described above can be improved by handling specially loop-invariant predicates. If the disjunction of the predicates guarding all the loads and stores of a same location contains a loop-invariant subexpression, then the initialization load can be lifted out of the loop and guarded by that subexpression. Consider Figure 4.48 on which we apply loop-invariant scalar-promotion.

By applying our register promotion algorithm one gets the result in Figure 4.49. However, using the fact that `c1` and `c2` are loop-invariant the code can be optimized as in Figure 4.50. Both Figure 4.49 and Figure 4.50 execute the same number of loads and stores, and therefore, by our optimality criterion, are equally good. However, the code in Figure 4.50 is obviously superior.

We can generalize this observation: the code can be improved whenever the disjunction of all condi-

```
for (i) {
    if (c1) *p += 1;
    if (c2) *p += 2;
    if (f(i)) *p += i;
}
```

Figure 4.48: *Sample code with loop-invariant memory accesses.* `c1` *and* `c2` *stand for loop-invariant expressions.*

tions guarding loads or stores from `*p` is weaker than some loop-invariant expression (even if none of the conditions is itself loop-invariant), such as in Figure 4.51. In this case the disjunction of all predicates is `f(i)||!f(i)` which is constant "true." Therefore, the load from `*p` can be unconditionally lifted out of the loop as shown in Figure 4.52.

In general, let us assume that each statement s is controlled by predicate with $P(s)$. Then for each promoted memory location `a[i+`j`]`:

1. Define the predicate $P_j = \vee_{s_j} P(s_j)$, where $s_j \in \{$statements accessing `a[i+`j`]`$\}$.

2. Write P_j as the union of two predicates, $P_j^{inv} \vee P_j^{var}$, where P_j^{inv} is loop-invariant and P_j^{var} is loop-dependent.

3. In prelude initialize `t`$_j$`valid = `P_j^{inv}.

4. In prelude initialize `t`$_j$ `= t`$_j$`valid ?  a[i`$_0$`+`j`] :  0`.[14]

5. The predicate guarding each statement s_j is strengthened: $P(s_j) := P(s_j) \wedge \neg P_j^{inv}$.

Our current implementation of this optimization in CASH only lifts out of the loop the disjunction of all predicates which are actually loop-invariant.

4.6.7 Discussion

The scalar promotion algorithm presented here is optimal with respect to the number of loads and stores executed. But this does not necessarily correlate with improved performance for three reasons. First, it uses more registers, to hold the scalar values and flags.

Second, it contains more computations than the original program in maintaining the flags. The optimized program may end-up being slower than the original, depending, among other things, on the frequency with which the memory access statements are executed and whether the predicate computations are on the critical path. For example, if none of them is executed dynamically all the inserted code is overhead. In practice profiling information and heuristics will be used to select the loops which will most benefit from this transformation. As the processor width and the cost of memory operations both increase, this optimization becomes more valuable since the predicate computations can often be scheduled in the available instruction slots.

[14] `i`$_0$ is the initial value of `i` in the loop.

```
/* prelude */
tmp_valid = false;

for (i) {
    /* first load from *p */
    if (! tmp_valid && c1) {
        tmp = *p;
        tmp_valid = true;
    }

    /* first store to *p */
    if (c1) {
        /* tmp_valid is known to be true */
        tmp += 1;
        tmp_valid = true;
    }

    /* second load from *p */
    if (! tmp_valid && c2) {
        tmp = *p;
        tmp_valid = true;
    }

    /* second store to *p */
    if (c2) {
        tmp += 2;
        tmp_valid = true;
    }

    fi = f(i);  /* evaluate f(i) only once */
    /* third load from *p */
    if (fi && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }

    /* third store to *p */
    if (fi) {
        tmp += i;
        tmp_valid = true;
    }
}

/* postlude */
if (tmp_valid)
    *p = tmp;
```

Figure 4.49: *Optimization of the code in Figure 4.48 without using the invariance of some predicates.*

```
/* prelude */
tmp_valid = c1 || c2;
if (tmp_valid)
    tmp = *p;

for (i) {
    /* first load from *p redundant */

    /* first store to *p */
    if (c1)
        /* tmp_valid is known to be true */
        tmp += 1;

    /* second load from *p redundant */

    /* second store to *p */
    if (c2)
        tmp += 2;

    fi = f(i);

    /* third load from *p */
    if (fi && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }

    /* third store to *p */
    if (fi) {
        tmp += i;
        tmp_valid = true;
    }
}

/* postlude */
if (tmp_valid)
    *p = tmp;
```

Figure 4.50: *Optimization of the code in Figure 4.48 using the invariance of* `c1` *and* `c2`*.*

```
tmp = *p;
for (i) {
    fi = f(i);
    if (fi)
        tmp += 1;
    if (!fi)
        tmp = 2;
}
*p = tmp;
```

Figure 4.51: *Code with no loop-invariant predicates, but with loop-invariant memory accesses.*

Third, scalar promotion removes memory accesses which hit in the cache,[15] therefore its benefit appears to be limited. However, in modern architectures L1 cache hits are not always cheap. For example, on the Intel Itanium 2 some L1 cache hits may cost as much as 17 cycles [CL03]. Register promotion trades-off bandwidth to the load-store queue for bandwidth to the register file, which is always bigger.

4.6.7.1 Dynamic Disambiguation

Our scalar promotion algorithm can be naturally extended to cope with a limited number of memory accesses which cannot be disambiguated at compile time. By combining dynamic memory disambiguation [Nic89] with our scheme to handle conditional control flow, we can apply scalar promotion even when pointer analysis determines that memory references interfere. Consider the example in Figure 4.53: even though dependence analysis indicates that `p` cannot be promoted since the access to `q` may interfere, the bottom part of the figure shows how register promotion can be applied.

This scheme is an improvement over the one proposed by Sastry [SJ98], which stores to memory all the values held in scalars when entering an un-analyzable code region (which in this case is the region guarded by `f(i)`).

4.6.8 Experimental Evaluation

In this section we present measurements of our register promotion algorithm as implemented in CASH. We show static and dynamic data for C programs from three benchmark suites: Mediabench, SpecInt95 and Spec CPU2000.

Table 4.5 shows how often scalar promotion can be applied. Column 3 shows that our algorithm found many more opportunities for scalar promotion that would not have been found using previous scalar promotion algorithms (however, we do not include here the opportunities discovered by PRE). CASH carries a simple flow-sensitive intra-procedural pointer analysis. The lack of precise pointer information, which is inherent in many C programs, is the greatest source of inhibition for register promotion [LC97].

Figure 4.54 and Figure 4.55 show the percentage decrease in the number of loads and stores respectively that result from the application of our register promotion algorithms. The data labeled **PRE** reproduces the data from Section 4.4.4, i.e., indicate the number of memory operations removed by our straight-line code

[15] Barring conflict misses within the loop.

```
tmp = *p;
tmp_valid = true;
for (i) {
    fi = f(i);

    /* first load */
    if (fi & !tmp_valid)
        tmp = *p;

    /* first store */
    if (fi) {
        tmp += 1;
        tmp_valid = true;
    }

    /* second store */
    if (! fi) {
        tmp = 2;
        tmp_valid = true;
    }
}
if (tmp_valid)
    *p = tmp;

-------------------------

tmp = *p;
for (i) {
    fi = f(i);
    if (fi)
        tmp += 1;
    if (!fi)
        tmp = 2;
}
*p = tmp;
```

Figure 4.52: *Optimization of the code in Figure 4.51 using the fact that the disjunction of all predicates guarding* `*p` *is loop-invariant (i.e., constant) "true" and the same code after further constant propagation.*

```
for (i) {
   s += *p;
   if (f(i)) *q = 0;
}

-------------------------------------

tmp_valid = false;
for (i) {
   if (!tmp_valid) {
      tmp = *p;
      tmp_valid = true;
   }
   s += tmp;
   if (f(i)) {
      *q = 0;
      if (p == q)
         /* dynamic disambiguation */
         tmp_valid = 0;
   }
}
```

Figure 4.53: *Code with ambiguous dependences and its non-speculative register promotion relying on dynamic disambiguation.*

optimizations only. The data labeled **loop** shows the additional benefit of applying inter-iteration register promotion.

The most spectacular results occur for `124.m88ksim`, which has substatial reductions in both loads and stores. Only two functions are responsible for most of the reduction in memory traffic: `alignd` and `loadmem`. Both these functions benefit from a fairly straightforward application of loop-invariant memory access removal. Although `loadmem` contains control-flow, the promoted variable is always accessed unconditionally. The substantial reduction in memory loads in `gsm_e` is also due to register promotion of invariant memory accesses, in the hottest function, `Calculation_of_the_LTP_parameters`. This function contains a very long loop body created using many C macros, which expand to access several constant locations in a local array. The loop body contains control-flow, but all accesses to the small array are unconditional. Finally, the substantial reduction of the number of stores for `rasta` is due to the `FR4TR` function, which also benefits from unconditional register promotion.

Our implementation does not use `dirty` bits and therefore is not optimal with respect to the number of stores (it may, in fact, incur additional stores with respect to the original program). However, dirty bits can only save a constant number of stores, independent of the number of iterations. We have considered their overhead unjustified.

Program	Variables				Program	Variables			
	Invariant		Strided			Invariant		Strided	
	old	new	old	new		old	new	old	new
adpcm_e	0	0	0	0	099.go	40	53	2	2
adpcm_d	0	0	0	0	124.m88ksim	23	10	1	4
gsm_e	1	1	1	0	129.compress	0	0	1	0
gsm_d	1	1	1	0	130.li	1	0	1	1
epic_e	0	0	0	0	132.ijpeg	5	1	9	5
epic_d	0	0	0	0	134.perl	6	0	0	1
mpeg2_e	1	0	1	0	147.vortex	22	20	1	0
mpeg2_d	4	3	0	0	164.gzip	20	0	1	0
jpeg_e	3	0	7	5	175.vpr	7	2	0	0
jpeg_d	2	1	7	5	176.gcc	11	40	5	2
pegwit_e	6	0	3	1	181.mcf	0	0	0	0
pegwit_d	6	0	3	1	197.parser	20	3	3	5
g721_e	0	0	2	0	254.gap	1	0	18	1
g721_d	0	0	2	0	255.vortex	22	20	1	0
pgp_e	24	1	5	0	256.bzip2	2	2	8	0
pgp_d	24	1	5	0	300.twolf	1	2	0	0
rasta	3	0	2	1					
mesa	44	4	2	0					

Table 4.5: *How often scalar promotion is applied. "New" indicates those cases which are enabled by our algorithm. We count the number of different "variables" to which promotion is applied. If we can promote arrays* a *and* b *in a same loop, we count two variables.*

4.7 Dataflow Analyses on Pegasus

As we have discussed in Section 4.3, Pegasus summarizes many important dataflow facts through its wires: def-use chains are represented by the data wires and may-dependences by the token edges. The multiplexors and mu nodes make Pegasus an SSA form. This information enables the compiler to perform many powerful simplifications without performing a dataflow analysis first.

However, some other optimizations do require the computation of dataflow information. The sparse def-use nature of Pegasus makes computation of such information very fast and easy. As already noted in [JP93], explicit dependence information makes dataflow computation almost trivial, since all that needs to be implemented is an abstract transfer function for each operation in conjunction with an iterative analysis. Since there is no control flow (or rather, the control-flow is explicitly encoded within the data-flow), there is no need for complex interval or structural analyses (although equivalents of these could certainly be used).

The forward dataflow algorithm can be expressed very succinctly as follows: for an operation with n inputs i_k and transfer function t, the output dataflow o can be written as a function of the input dataflow facts $i_k, k \in \{0, \dots, n\}$ as: $o = t(i_0, i_1, \dots, i_n)$. The multiplexors and mu are the join points, whose output is given by $o = \sqcup_k i_k$ (for k ranging over the data inputs, i.e., excluding predicate inputs for the mux).

A backward analysis is just slightly more complicated: if output o fanouts to inputs $o_k, k \in \{0, \dots, n\}$, then $o = \sqcup_k o_k$ and $i_k = t_k^{-1}(o)$ (i.e., the reverse transfer function for the k-th input).

We will illustrate dataflow analysis by giving three example optimizations used in CASH. All these

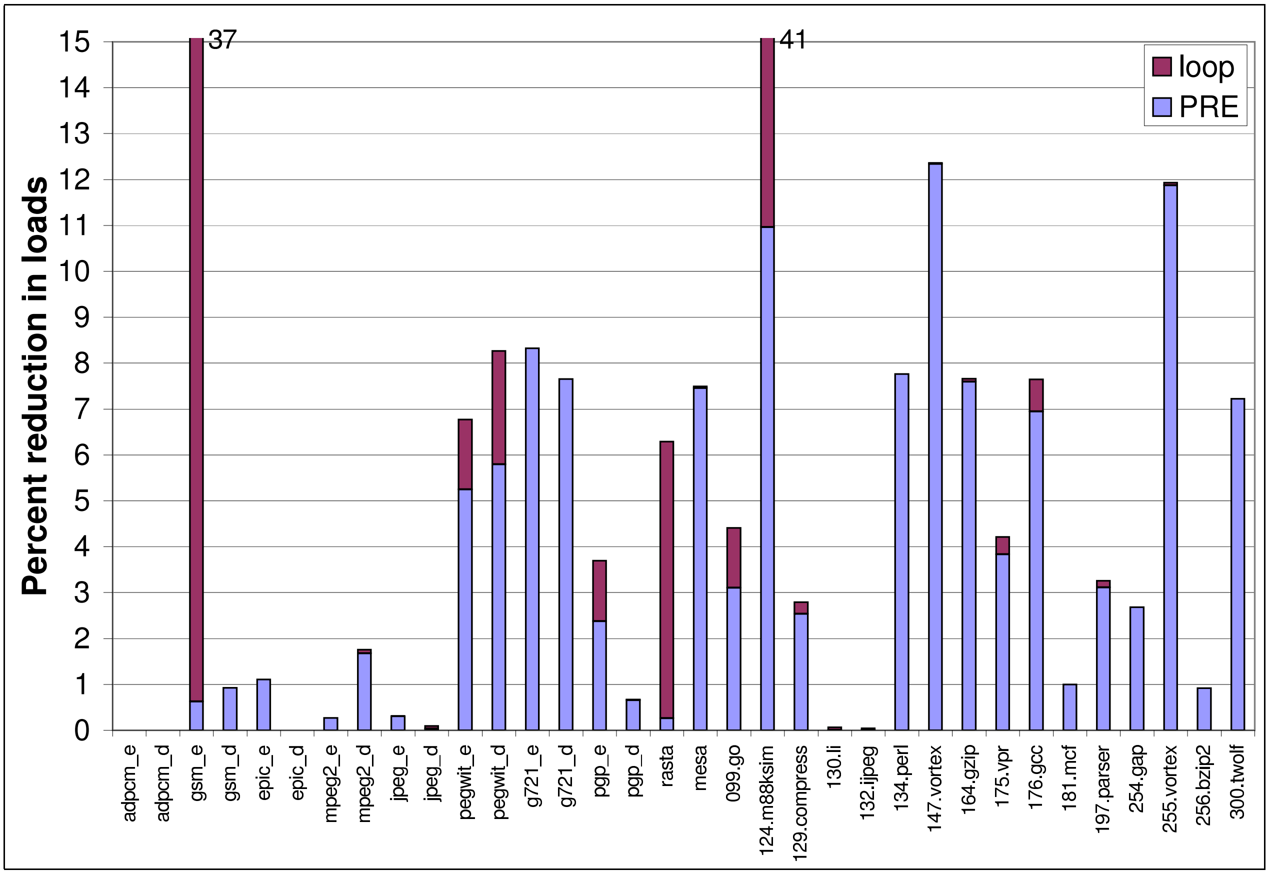


Figure 4.54: *Percentage reduction in the number of dynamic load operations due to the application of the memory PRE and register promotion optimizations.*

analyses are global at the procedure level. We present constant propagation, a forward analysis, dead-code, a backward analysis on the scalar subgraph only, and dead local variable removal, a backward analysis which operates solely on the token-induced subgraph.

For all these problems the lattice depth is constant and the dataflow computation is linear-time. The implementation is also extremely succinct. For example, the *complete* implementation of the dead-local variable removal from Section 4.7.3, encompassing both the dataflow analysis and the actual removal of the dead operations, is less than 100 lines of source code with comments and spaces (Table 4.1).

We plan to add more dataflow analyses to Pegasus. For example, we want to re-write the BitValue analysis [BSWG00], which discovers useless bits. Implementing the analysis on Pegasus ought to be straightforward. The results produced by BitValue ought to benefit from the superior register promotion in Pegasus, since BitValue acts very conservatively at memory operations.

4.7.1 Global Constant Propagation

Global constant propagation is a very simple forward dataflow analysis which replaces some "variables" with constant values. (It should not be confused with constant folding, which computes the output of side-effect operations having all inputs constant.) In Pegasus constant propagation is almost trivial, involving exactly four operations: noops, hold, mu and eta. Since there are no variables in Pegasus, only wires,

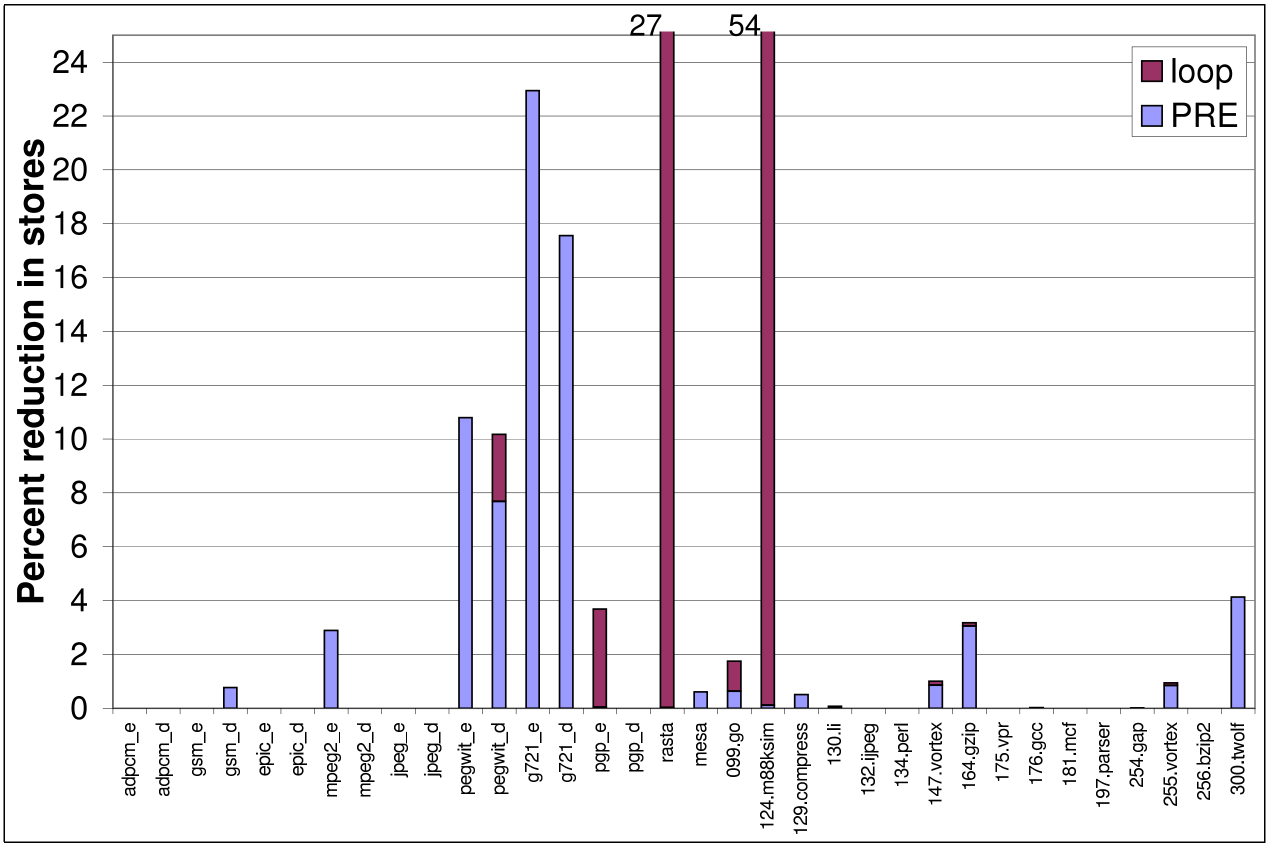


Figure 4.55: *Percentage reduction in the number of dynamic store operations due to the application of the memory PRE and register promotion optimizations.*

Operation	**Initial value**
Constant c	c
Any other	$\top$

(A)

Operation	**Transfer function**	**Comments**
Noop	$o = i$	Only one input
Eta, hold	$o = i_0$	The data input
Mu	$o = \sqcup_k i_k$	For all data inputs i_k

(B)

Table 4.6: *(A) Initializations and (B) dataflow transfer functions for the global constant-propagation problem.*

copy propagation is automatically performed when connecting definers to users. What used to be “copy” instructions become nothing, or perhaps, under some conditions, noops.

The lattice is the standard one, containing all constants, not comparable to each other, a $\perp$ element which is “less than” any constant, and a $\top$ element, which indicates that a value is not reachable. The initial values and transfer functions for all operations are given in Table 4.6, After performing the dataflow analysis an operation having the associated dataflow output a constant can be replaced with that constant.

Operation	Initial value
Store, call	$\top$
Frame, pop	$\top$
Return, arg	$\top$
Any other	$\bot$

(A)

Operation	Transfer function
Any	$\forall k.i_k = \sqcap_j o_j$

(B)

Table 4.7: *(A) Initializations and (B) dataflow transfer functions for the dead-code problem.*

Operation	Initial value
Delete	$\bot$
Any other	$\top$

(A)

Operation	Transfer function	Comments
Store	$i = o$	
Create, delete	$i = o$	
Combine, noop, eta, mu	$i = o$	
Load, call, frame, pop	$i = \top$	Never dead

(B)

Table 4.8: *(A) Initializations and (B) dataflow transfer functions for the dead local variable problem. The only inputs and outputs considered are the token i/o's.*

4.7.2 Dead Code Elimination

Global dead-code elimination is a very simple backwards dataflow analysis used to discover operations whose output is unused. It operates solely on the scalar values in Pegasus, ignoring the token edges altogether. The used lattice is trivial, containing just two values, "useful" $\top$ and "dead" $\bot$. Table 4.7 shows the initialization and transfer function: side-effect operations which cause mutations are initialized to $\top$, and all other to $\bot$. The dataflow fact of all inputs is the $\sqcap$ of all output (wires) of an operation, and each output is the $\sqcap$ of all its destinations.

After dead-code an operation having the associated dataflow output $\bot$ can be removed from the graph. Operations having tokens need to connect the token input to the output on removal.

4.7.3 Dead Local Variable Removal

Finally we discuss a backward dataflow analysis used to discover dead local variables whose address is taken and that can be removed from the program. Even if not removing the variables entirely, this analysis can remove "dead stores", which are performed to the variable just prior to exiting the scope in which the variable is live. In conjunction with register promotion this optimization can sometimes produce quite spectacular results. This optimization translates the circuit in Figure 3.17 to the one in Figure 3.18.

This dataflow analysis needs only to consider the token edges of the Pegasus graph, since it is an analysis of side-effects. The analysis is therefore defined only operations with token inputs and outputs. Its computation is described by Table 4.8. Basically, each store to a variable immediately followed (only) by a *delete* or another dead operation is dead. A *create* in the prologue followed only by dead operations is dead itself, and therefore can be removed from the graph, together with the corresponding *delete* in the epilogue. Since this is a backward problem, an output needs to perform a $\sqcap$ on all its destinations before applying the transfer function. In fact, this is the only way that a value can go "up" in the lattice.

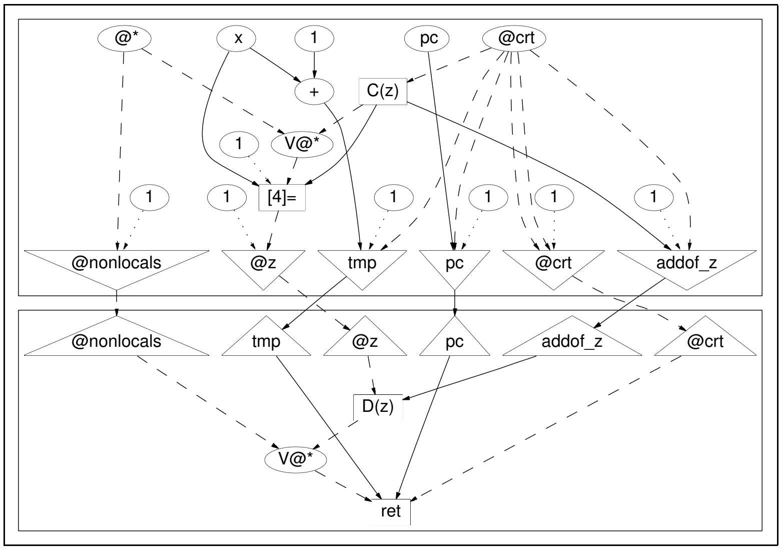


Figure 4.56: *The program in Figure 3.17 after register promotion and just before the dead-local variable removal.*

Going back to Figure 3.17, let us see how this analysis operates. First, the store-load forwarding transformation shown in Figure 4.18 removes the redundant load from `z`, generating the result in Figure 4.56. Then, the "deadness" propagates as follows along token back-edges: `Dz` ← mu `@z` ← eta `@z` ← `[ ]=` ← `V@*` ← `Cz`. First, the `[ ]=` store is certainly dead and can be removed anyway. Next, since the *create* is also dead, it can be removed together with the corresponding *delete*, leading to the result in Figure 3.18.

Chapter 5

Engineering a C Compiler

This chapter contains some of the software engineering principles we have quite painfully discovered while belaboring at CASH and its related infrastructure. We felt compelled to include this advice here for three reasons:

- We think it is useful. In fact, had we known and applied some of these principles early, we would have saved some time and, much more importantly, frustration.
- In our course of study and research we have read too little on these sort of techniques. In fact, we think that systems engineering education should contain at least some modicum of code-writing advice.
- There is no other place where we can write about this knowledge. It is too sparse and disorganized for a real publication and too hard to quantify from a scientific point of view.

As any coding style guide, some of the preferences here will be found distasteful by some readers. Also, some of the techniques described will be mostly suitable for the specific task of writing compilers. These, unfortunately, are a particularly difficult subject of development, since bugs often manifest themselves as second-order effects only: the compiler output is a legal program, but when run it produces incorrect results. Finding the source of the problem, often *without understanding* the compiled program itself, is an exercise in patience and strong will. Therefore, here is a bag of tricks, in no particular order.

5.1 Software Engineering Rules

Determinism: Every run of the program with the same input should produce the same output and go through the same execution path. Non-determinism should be confined to as small a piece of code as possible. In CASH, whose central data structure is the hash table, pointers to various objects are hashed according to their actual values. Hash table iterators used to scan the hash table always from top to bottom. But the location of an object in the hash would vary according to its address, which could depend, for example, on the environment of the process. The fix, implemented after many frustrating months, was to add to each hash table a doubly-linked list holding the hash elements in the insertion order. The list is used by all hash iterators, while the regular hash is used for searches. This makes the hash somewhat slower (but still amortized constant time per operation), but deterministic.

Correctness first: This is a well known piece of advice, of particular value for research code: *premature optimization is the root of all evil*. In other words, always prototype with the simplest algorithm you

```
         /* incorrect */

void optimize(Hyperblock* h)
{
    ...
    transitive_reduction(h);
    pre_for_memory(h); /* needs transitive reduction */
}

-----------------------------------------------------
           /* correct */

void pre_for_memory(Hyperblock* h)
{
    transitive_reduction(h); /* required invariant */
    ...
}
```

Figure 5.1: *Each optimization pass establishes all the invariants required on its own.*

can, even if its complexity is inferior. Proceed to replace it only if it proves to be the bottleneck. Use profiling (e.g., `gcc -pg`) to find the actual bottlenecks.

Clean semantics: In Pegasus the semantics of each operation is precisely defined. The program is also at all points strongly typed and consistent. Even a small ambiguity can make life much harder, since program invariants are much harder to reason about.

Freshness: CASH does not rely on the results of an analysis or transformation carried in the past because the program may have changed since. If CASH needs to establish an invariant on the data structure (e.g., transitively reduced token graph), it explicitly invokes the computation before being sure that the invariant is correct, see Figure 5.1. Too often not doing this led to painful bugs when compilation passes were reordered.

Caching: If some information can be cached, in CASH the caching is always clearly explicit. For example, since computing reachability is expensive, CASH uses a memoization table. This table is not a permanent object. It is allocated by the caller of the reachability function and deallocated by the caller, as illustrated in Figure 5.2.

Invariants: Related to caching: if the information you want to cache is expensive to compute and you would rather update it incrementally, then make it an *explicit invariant* of the data-structure and maintain it carefully in all transformations. Or ensure that it is recomputed only after batches of other transformations.

Modular transformations: It is very useful if optimizations can be reordered easily, to allow with experimentation.

```
           /* incorrect */

static CACHE dependence_cache;

bool dependence(Node* n1, Node* n2)
{
    if (dependence_cache.member(n1, n2))
        return dependence_cache.lookUp(n1, n2);

    ...
    dependence_cache.cache(n1, n2, retval);
    return retval;
}

-------------------------------------------------------------
           /* correct */

bool dependence(Node* n1, Node* n2, CACHE *dependence_cache)
{
    if (dependence_cache && dependence_cache->member(n1, n2))
         return dependence_cache.lookUp(n1, n2);

    ...
    dependence_cache->cache(n1, n2, retval);
    return retval;
}

void pre_for_memory(Hyperblock* h)
{
     CACHE dependence_cache;
     /* at this point the cache is empty */

     ForAllTokenEdges (e) {
         if (dependence(e->node1, e->node2, &dependence_cache))
            ...
     }
}
```

Figure 5.2: *The lifetime of caching of intermediate results is explicitly controlled.*

5.2 Design Mistakes

Here are some important design flaws in CASH. Planning for long-term evolution of a compiler should have avoided them.

- Outer loops are not first-class objects. Optimizing them in the current framework is not easy.

- Hard-coded C memory model. The current compiler is not easy to extend to handle, for instance, FORTRAN programs.

- Single procedure at a time. CASH relies on some `static` data structures to represent the current procedure. This means there cannot be several procedures optimized simultaneously.

- Not portable. CASH relies in many places on the native arithmetic and has some hard-coded dependences on 32-bit integers.

- Hard-coded limitations. The fact that an operation cannot have more than two outputs was deeply embedded in the code. When we realized that the call operation needs three outputs, we had to resort to ugly workarounds. Future extensions may be jeopardized by this decision.

5.3 Debugging Support

Debugging is never too easy. Whatever tools one can create to simplify this task usually pay off in the long run. A robust debugger is a must, but coding for debugging is also necessary.

- Very valuable is the ability to turn each optimization on and off in the compiler, either globally or even on a per-function basis. Inhibiting optimizations one by one is useful when trying to discover the source of a bug.

- Very useful is the ability to selectively optimize only some of the functions in the input program. We use a Perl script to do a binary search to narrow down the location of a bug.

- The liberal use of `assert()` calls pays off handsomely in the long run.

- Functions to check major invariants of the data structures are invoked repeatedly at run-time. Command-line debugging options can be used to turn on more of these.

- Each internal object has a unique associated ID; the ID is generated in a single function (even if the class has several different constructors). The ID is invaluable for debugging. If you need to know when an object is created, use a conditional breakpoint in the ID initialization function.

- A computer is fast, so let it do most of the work in finding or explaining a bug. For example, when an invariant is broken, the compiler may try to summarize the portion of the data structure with the problem. For example, if an optimization introduces an illegal cycle in Pegasus, topological sorting will discover it and will print it as a list of unique IDs.

- Compilers use highly complex data structures. A visualization tool is invaluable. We have used both `dot` [GN99] and `vcg` [LS93]. The latter produces less refined layouts, but is faster to run and can display more information. It is very important that the compiler be able to dump the data structures for visualization *even if the structure is incorrect* (i.e., it breaks invariants). The graph dumping routines should therefore be as robust as possible. They can be invoked from the debugger command-line at various points during the compilation process to see a snapshot of the current state of the representation. The visualization tool ought to display for each object the unique ID, as shown for example in Figure 4.9, where IDs are between square braces. Various options should control the amount of detail. In `dot` we use color to supplement information which is not controllable through the layout, e.g., the input number of each wire.

- All optimizations can be made to print a trace of the applied transformations when command-line debugging flags are turned on.

- Once a compiler bug is detected, a simple piece of input code triggering it is added to a regression test suite. Frequently later changes may exhibit the same bug, now quickly caught by the test.

- When compiling large functions, or when using aggressive unrolling or inlining circuits can grow very large. A collection of Perl scripts was used to trim down the size of the graph printouts. Most often these tools are used in an interactive way: analyze graph, display more nodes, repeat.

- We found it very useful to institute run-time tests against the most egregious coding errors in the compiler. For example, we noticed that repeatedly we were modifying a data structure somewhere deep in a call chain, while a caller had an iterator over the same data structure. Unless the changes are very restricted, the iterator will point to garbage on return. Therefore we have guarded all data structures with a flag showing whether they are currently being iterated upon. All methods modifying the data structure check this field first and immediately abort if an iterator is in use. This has uncovered a surprising number of very shallow bugs.

- A solid regression testing suite is hugely useful since it allows debugging with small inputs.

- Having an automatic way to visualize the program evolution *at run-time* is a huge time-saver. More details about the tools we have developed are in Section 6.3 and Section 6.6.

- Analyzing the code coverage of the compiler (e.g., by using `gcc -ftest-coverage`) can sometimes point to the need for additional regression tests, or even subtle bugs.

5.4 Coding in C++

Finally, this section presents some language-related considerations. CASH is built in C++ and we mention here some of the ways we found the language useful or cumbersome.

- CASH does not use the exception mechanism in C++.

- CASH does not use inheritance at all. Instead of virtual dispatch functions, we prefer to use large switch statements. The resulting code is easier to understand and maintain and is faster as well. (This way of structuring code resembles somewhat Suif's organization.)

- CASH makes heavy use of templates. 99% of CASH is type-safe (i.e., uses no wild pointer casts), and this is due to the heavy use of templates. Unfortunately, at least with `gcc`, templates cause huge code blow-ups and are difficult to debug (because there is no way to put a breakpoint just for `Hash<int, int>`).

- When CASH was started, the Standard Template Library (STL) was too immature to be trustworthy, therefore CASH does not use it at all. Once we had developed a good library of data structures (hashes, lists, arrays), we found the need for STL greatly diminished.

- We found it more useful most of the time to organize code in files according to functionality and not to class. E.g., all term-rewriting optimization functions for all classes are in a single file.

- In retrospect, we have made too many optimization routines members of the `Circuit` class (i.e., the main Pegasus structure corresponding to a hyperblock). We should have paid more attention to Suif, which has an excellent code organization. Only important accessors should be methods, and optimizations should be global functions, receiving mutable `Circuit*`'s as arguments.

- Finally, the main data structure in CASH is the (polymorphic) hash table. Hashes are resized automatically to keep the expected access cost constant. Hashes are frequently used instead of creating new object fields.[1] For example, if we need to find all `Node`s with a certain property, we build a hash mapping `Node`s to properties and populate it when computing the property. The alternative would be to add a new field to the `Node` class for each property, making the class quickly very baroque. Suif 1.x itself relies too much on linked lists and uses almost no hashes internally. Some of the speed problems of Suif are clearly attributable to lists. The amortized constant-time access to a hash makes it almost as fast to access as a true object field.

[1] Ironically, hashes are the hardest structures to implement effectively in Spatial Computation.

Part II

ASH at Run-time

Chapter 6

Pegasus at Run-Time

The first part of this thesis was about the static aspects of Spatial Computation. Now we turn our attention to its run-time behavior. In this chapter we study the general behavior of a spatial implementation at a local and global level. We discuss a series of optimizations which are uniquely applicable for spatial asynchronous circuits and then analyze in detail one of the main vehicles of high performance, namely dataflow software pipelining. The next two chapters are devoted to analyzing the performance of the synthesized circuits: Chapter 7 applies CASH on embedded systems benchmarks, while Chapter 8 evaluates the effectiveness of CASH on control-intensive pointer programs, contrasting the behavior of Spatial Computation with superscalar processors.

6.1 Asynchronous Circuit Model

In this section we detail the protocol of the nodes at the ends of a “wire” connecting a producer operation with multiple consumers. Figure 6.1 is an expansion of Figure 3.1 from Section 3.2 for the case of multiple consumers.

Each operation has an associated output latch,[1] used to preserve the output value as long as required for all consumers to read it.[2] An operation will not overwrite the output value if there is at least one consumer that has not acknowledge the receipt of the prior output. Handling multiple outputs only requires a supplemental “and” gate to combine the acknowledgements of all consumers.

There is a lot of research in asynchronous circuits on improving the signaling protocols between producer-consumer operations (e.g., [Wil90, SN00, CNBE03]). However, this text uses for all simulations a relatively simple communication model. The behavior of practically all operations is described by the same finite-state machine.

6.1.1 The Finite-State Machine of an Operation

Figure 6.2 shows a representation of the Finite-State Machine (FSM) governing the behavior of each node at run-time. The FSM is broken into three interdependent small FSMs governing the behavior of the inputs, computation and outputs respectively. An operation having multiple outputs will have a single input FSM, but a copy of the data and output FSM for each of its outputs. Transitions of these FSMs are not always

[1] Depending on the actual hardware implementation, a real latch may be unnecessary. For example, dynamic asynchronous logic provides a latching effect without using a flip-flop [SN00].

[2] However, remember that some operations have multiple outputs; for example, loads generate both a data value and a token.

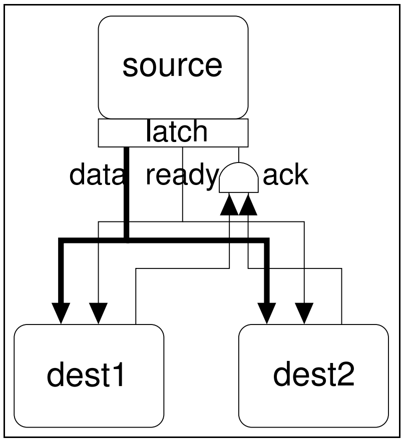


Figure 6.1: *Signaling protocol between data producers and consumers.*

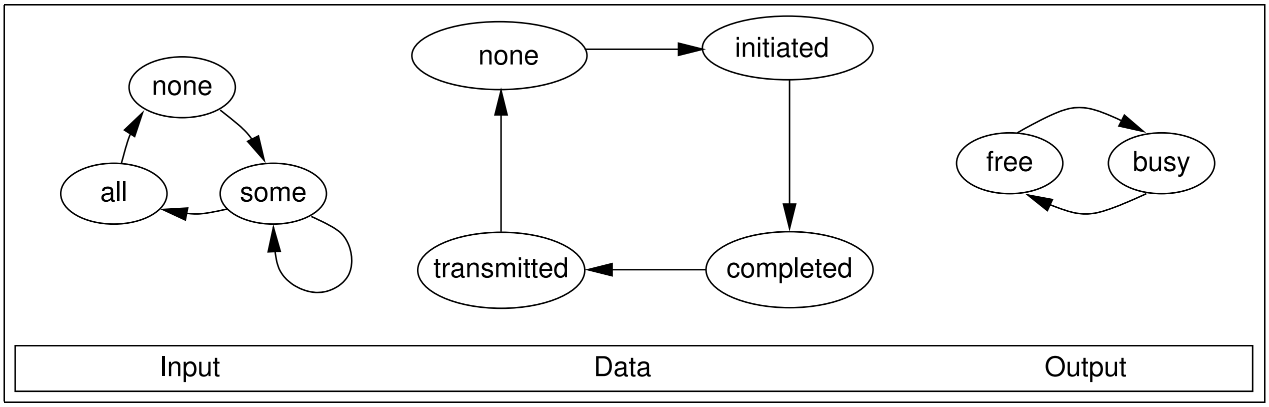


Figure 6.2: *Finite-State Machine describing the computation of an asynchronous node.*

independent. The following description shows the events that trigger the transitions and also the events triggered by each transition:

Output

free → **busy** is a transition that occurs when an output is produced on the output wire

busy → **free** occurs when all acknowledgments have been received for the output

Input

none → **some** occurs when first input data is received

some → **some** occurs when one more input has been received (but not for the last input)

some → **all** occurs when the last input data is received

all → **none** occurs only when the output has been latched and an acknowledgment has been sent for all inputs

Data

none → **initiated** taken when enough inputs have arrived to enable the computation of the output value — the computation is started

initiated → **completed** taken when the computation of the output value is completed (but output is not yet latched)

completed → **transmitted** taken after all acknowledgments have been received from consumers

transmitted → **none** taken after operation has sent all acknowledgments

Although this FSM looks quite complicated, its hardware implementation is quite simple (and fast) by using just two asynchronous C gates for each operation, in the style of micropipelines [Sut89].

The main point of this elaborate description is that, due to the presence of the output latch, an operation can start computing even if the previous output it has produced has not been consumed yet. Because the operation will not acknowledge its inputs before latching the output, this behavior is safe even if the latch is "busy" with the previously computed value. Another important point is that an operation acknowledges *all* its inputs at once, even if it requires only some of them to compute the output (i.e., it is lenient as described in Section 6.4). This last step is required for a correct implementation. As an illustration, if each input would be acknowledged separately, an "and" operation could receive two consecutive "0" values of the left input before having received the right input at all.

6.2 Timing Assumptions

While the implementation generated by CASH can be a truly asynchronous circuit without any notion of global clock, for simulation purposes we take the simplifying assumption that all delays are integer multiples of a basic time unit, which we will call from now on "clock cycle." We assume that computation time dominates propagation delays on the wires, which are approximated to zero in the forward direction. However, propagation of the acknowledgment signal takes one clock cycle.

The actual delays will depend on a variety of factors, including technology mapping, synthesis back-end and circuit layout. The Verilog models in Section 7.5 use precise measurements for evaluating performance. However, for most of the evaluations in Chapter 7 and Chapter 8 we assume that for most operations the latency (measured in clock cycles) in Spatial Computation is the same as in a superscalar processor. We also tacitly assume the same clock cycle for both implementations. Since processors are highly tuned, it is not clear how realistic this assumption may be. This timing model enables us to focus on the fundamental performance of the computational model, ignoring the impact of technology.

6.3 State Snapshots

In order to better understand the properties of ASH we will take a look at some possible snapshots of a Spatial Computation. We further elaborate the analysis with a discussion of dataflow software pipelining in Section 6.7.

In order to better understand the behavior of the generated circuits we have developed a relatively crude but effective visualization tool. The tool is based on three ingredients: (1) circuit drawings, generated by CASH using the `dot` back-end; (2) a circuit simulator, which can be configured to dump traces of important events and FSM transitions; and (3) a Perl script which combines the two, by repeatedly marking operations with color codes corresponding to their state. A sample simplified trace (due to the limitation of this document to black and white) is shown in Figure 6.6 below.

As with superscalar processors, the state of the computation is quite complex, and even more so, since Spatial Computation has neither a program counter nor a finite-size instruction window: at any one moment in time there may be operations which have received some of the inputs, operations which are currently computing the result, operations blocked waiting for an acknowledgment, and idle operations. An important invariant is that once "execution" enters a hyperblock, all operations within that hyperblock are executed exactly once. If the hyperblock is a loop body, execution enters the hyperblock repeatedly, and for each entry all operations must go through a complete computation cycle: get all inputs, produce output, receive acknowledgment.

Within loops or recursive functions some operations may receive a new round of inputs before having computed the previous output. Such nodes will start immediately computing a new output value but will be able to latch it and pass it to the successors, only when all outputs have acknowledged the prior value. There is no way a "later" value can overtake an "early" one in the computation.

In order to better understand how execution proceeds, it is useful to define a "generalized iteration index" (which we will call simply "index"), which generalizes the notion of iteration index vector [KKP$^+$81] often used to analyze loop traversal orders in parallelization transformations. Unlike iteration index vectors, which are a per-nested-loop notion, the index is a per-program notion. The index is a positive integer, incremented whenever execution crosses from a hyperblock to a successor hyperblock. The index can be also seen as an abstract form of the tag used in the tagged token dataflow machine [AN90]. We should think of each data item being tagged with its index. Whenever data goes along a cross-edge (i.e., a CFG edge spanning two different hyperblocks, or a back-edge, cf. Section 3.3), its index is incremented.

A very important run-time invariant is then the following:

> **For any operation, all inputs that contribute to the computation of the output have the same index.**

A second important invariant governs the order of the computations:

> **The consecutive values flowing on every edge have strictly increasing indices.**

If we freeze the computation at an arbitrary point in time and take a snapshot at the "live" values on wires (i.e., the ones that have data-ready set, but are not yet acknowledged), they can have widely differing indices. Also, live values can be present simultaneously in multiple hyperblocks, and even in multiple procedures, because a procedure invocation can proceed in parallel with the execution of independent operations in the caller. If lenient execution is employed (see Section 6.4 below), then there can be live values within a procedure even after the procedure has executed its return operation. Due to the implementation of recursion (see Figure 3.21), however, there can be no live values in two invocations of the same procedure simultaneously: the recursive call instruction "drains" all live values from a procedure in order to make the recursive call. We further revisit state snapshots in Section 6.7, where we take an in-depth look at the behavior of loops and at the emerging property of dataflow software pipelining.

6.3.1 Switches

In Appendix B we argue formally about the correctness of the above two invariants using an induction argument over the length of the execution. However, the use of *mu* operations as merge points for control flow cannot guarantee these properties. Figure 6.3 shows two circuits which may violate the above invariants. In (A) one data item iterates around the loop and reaches a mu before the value that was meant to initialize the mu before the start of the first iteration. In (B) while some data still circulates within the left side of

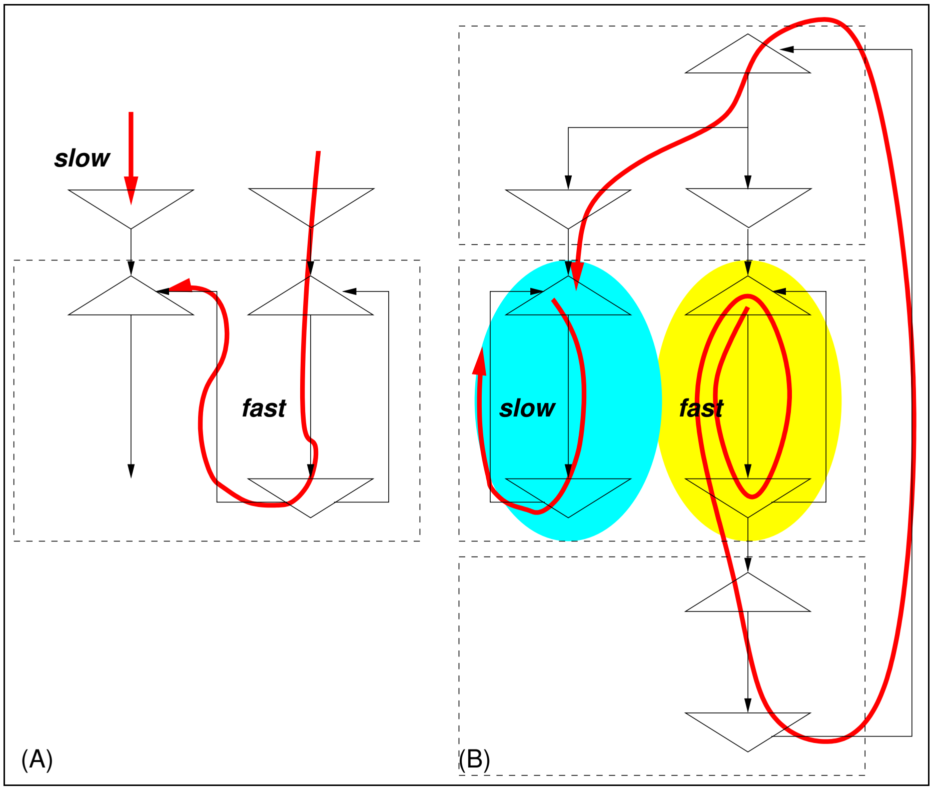


Figure 6.3: *Two circuits whose execution may violate correctness invariants.*

the innermost loop, other data can terminate all iterations on the right side of this loop, exit, go around the outermost loop and re-enter the innermost loop.

Intuitively this phenomenon happens because data can follow different paths to reach a mu. While on an edge all indices are strictly increasing because the data can reach the mu from different edges, the data with a larger index can reach the mu before the data with the smaller index.

To fix this problem we introduce *switch* nodes. Switches are a hybrid between mu and multiplexors. Like muxes, switches have a control input, which indicates *which* of the data inputs is expected next. Like mu operators, a switch does not expect to receive all its inputs, but only the one indicated by the control. When the indicated input arrives, it is forwarded to the output and both the control and data input are acknowledged. If any other input shows up, it is simply ignored (i.e., not acknowledged).

Replacing all mu operators with switches restores the invariants, assuming there is a reliable way to generate the control signal for selecting the correct input. For this purpose we use the mu operators for the *current execution point* (crt). When such a mu receives data on one input, it indicates to all switch operations to use the corresponding input as well. The crt mu operators are guaranteed to always receive the correct input because at any one point there is exactly one active crt value in the whole program. Figure 6.4 shows how the loop hyperblock in Figure 6.3(a) is modified by using switch nodes to steer the “mus.”

The “mu” node will indicate to both “switches” to accept their initialization inputs first. Although the “fast” value will travel quickly and reach the leftmost switch node first, it will remain blocked there, since the switch is open only for the “slow” value.

Note that our usage of “switch” nodes is subtly different from the traditional usage of switches in the dataflow literature [DM98], which requires the control predicate of a switch to be already present at the start

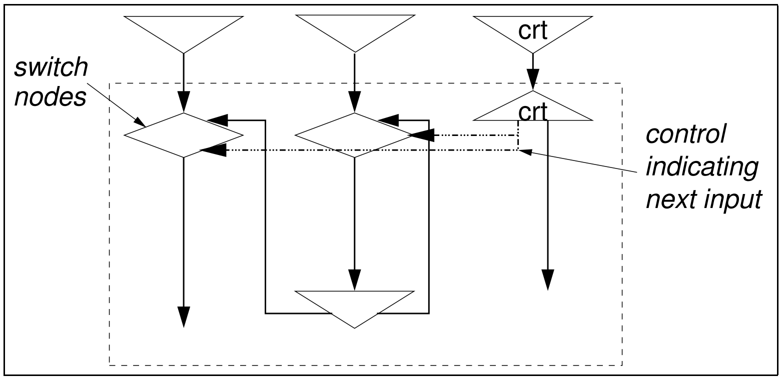
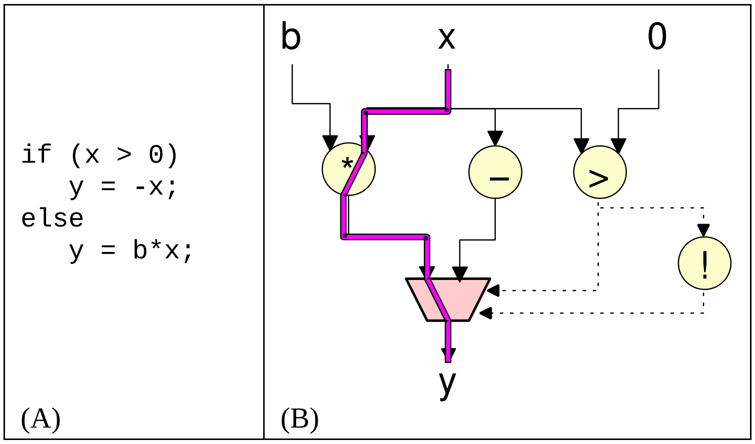


Figure 6.4: *Usage of "switch" operators inside controlled by the "mu" for the current execution point. The "mu" indicates to the switch nodes which of the inputs should be forwarded next.*



```
if (x > 0)
   y = -x;
else
   y = b*x;
```

Figure 6.5: *(A) Sample program fragment and (B) the corresponding Pegasus circuit with the static critical path highlighted.*

of the computation in order to allow data to enter the loop. Our scheme has two advantages: (1) it does not require any state when the computation is initiated (i.e., the FSM of each node starts in the same state) and (2) it works even for programs having irreducible CFGs.

6.4 Lenient Execution

As indicated in Section 3.3, Pegasus uses explicit multiplexors to represent the join of multiple definitions of a variable. Recall that hyperblocks are the basic unit of (speculative) execution: all operations within a hyperblock are executed to completion every time the execution reaches that hyperblock.

In the literature about EPIC processors, a well-known problem of speculation is that the execution time on the multiple control-flow paths may be different [AmWHM97]. Figure 6.5 illustrates the problem of balancing: the critical path of the entire construct is the longest (worst-case) of the critical paths of the combined conditional paths. Traditional software solutions use either profiling, to ensure that the long path is overwhelmingly often selected at run-time, or exclude certain hyperblock topologies from consideration,

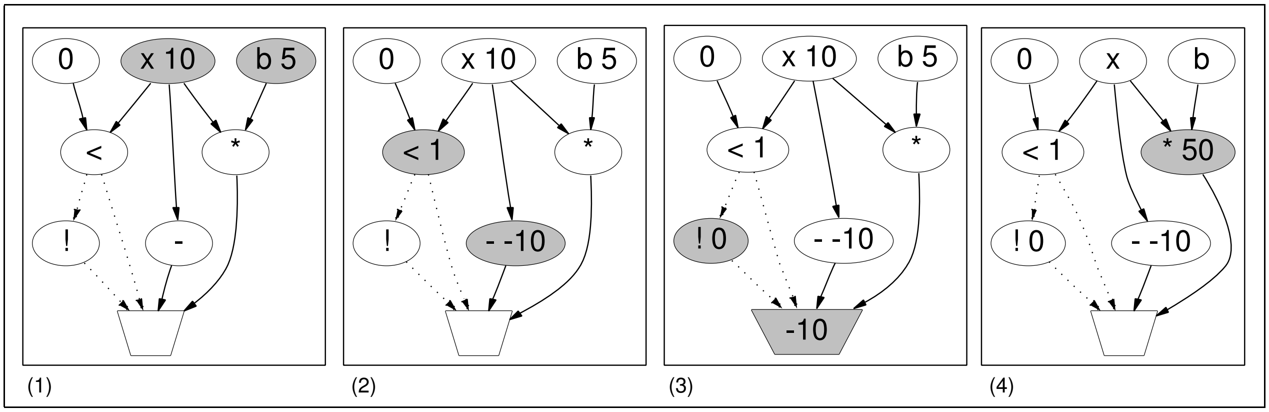


Figure 6.6: *Evaluation of the program in Figure 6.5 for inputs* `x=10` *and* `b=5`*. The shaded operations are currently executing. We have displayed in each node the output value. Each value is shown as long as it has not been acknowledged. Notice that the lenient multiplexor generates an output of* -10 *before having received the results from the negation and multiplication. Also, notice that the multiplexor receives the acknowledgment for its output before having sent its own for the inputs.*

rejecting the predication of paths which differ widely in length.

In Spatial Computation we can use a completely different method to solve this problem: we make multiplexors and predicate computation operations *lenient*. Leniency is a property of the execution of an operation, related to strictness and laziness. An operation is strict if it requires all its inputs to compute the output. An operation is non-strict if it can produce a result with just some of the inputs. For example, in most programming languages the `if` construct is non-strict, since it evaluates only one branch. A lenient operation is somewhere between strict and non-strict: it can produce the output with just some of the inputs, but nevertheless expects all the inputs to be present. Lenient operators generate a result as soon as possible, while still expecting all inputs to eventually become available. For example, a "logical and" operation can determine that the output is "false" as soon as one of its inputs is "false." However, lenient operations have defer the sending of acknowledgments until all of the inputs have been received. In order to understand this restriction, consider an "and" operation whose left input is "false:" if the input is acknowledged, the next value of the same input could arrive before the other inputs have been received. In the asynchronous circuits literature leniency was proposed under the name "early evaluation" [RTT01]. A recent enhancement of the protocol in [BG03a] relaxes the restriction of simultaneous acknowledgements.

Lenient evaluation should not be confused with short-circuit evaluation: a short-circuit evaluation of an "and" always evaluates the left operand, and if this one is "true", it also evaluates the right one. However, a lenient evaluation of "and" generates a "false" result as soon as *either* input is known to be "false." Short-circuit evaluation is a static property, while lenient execution is a dynamic property.

Multiplexors have lenient implementations as well: as soon as a selector is "true" and the corresponding data is available, a mux generates the output. Considering the circuit in Figure 6.5 again, let us assume that all inputs show up simultaneously: `x=10,b=5` (i.e., the multiplication result is not used). Figure 6.6 shows the evolution in time of the computation: the multiplexor emits the result of the subtraction as soon as it is available. The multiplication output is simply ignored.

As a result of leniency, at least for this sample circuit, *the dynamic critical path is the same as in a non-speculative implementation*. That is, if the multiplication output is not used, it does not affect the critical path. As shown in Section 6.7, the multiplication delay can still impact the dynamic critical path if it is within a loop, but this can be managed by pipelining the implementation of the multiplier.

Besides Boolean operations and multiplexors, there are other operations which are only "partially" lenient. For example, all predicated operations are lenient in their predicate input. For example, if a load operation receives a "false" predicate input, it can immediately emit an arbitrary output (by raising the "data ready" signal), since the actual output is irrelevant. Other operations could be made lenient in some weaker sense still. For example, a multiplication receiving a 0 input could immediately output a 0 result.

The use of leniency gives rise to an interesting run-time behavior, well illustrated by Figure 6.6. Let us assume that the mux is immediately followed by a "return" instruction. Then it is entirely possible for the return to be executed before the multiplication. Therefore control can return to the caller of a function even though there is still active computation carried on within the function. A second consequence is that execution has returned, but the multiplier has not yet acknowledged its inputs. A new call to the function containing this code may therefore be attempted before the values of the inputs `x` and `b` have been consumed. This points out the need to employ flow-of-control mechanisms for function call operations: a function may *not* be ready to accept new inputs, even though an old invocation of it has completed.

The multiplexors and eta nodes are natural *speculation drain* nodes: incorrectly speculated execution is automatically squashed at multiplexors or etas controlled by "false" predicates.

6.5 Memory Access

In Pegasus memory access is represented by load and store operations. As described in Section 3.2, both loads and stores not only have data and address signals, but also a predicate input and a token input and output. The predicate is required because memory operations have side-effects which cannot be un-done if wrongly speculated, unlike scalar speculation which is simply drained at multiplexors. The input and output tokens are used to enforce the execution order between potentially non-commutative memory operations which do not depend on each other through scalar values.

One of the radical differences between our proposed model of Spatial Computation and an ordinary microprocessor is the actual embodiment of the tokens in the "virtual ISA." In a microprocessor, memory access instructions are implicitly ordered according to the (total) program order. Run-time disambiguation may re-order memory instructions by comparing the addresses they access. But in case of dependence the ordering is enforced by the actual issue order. Since Spatial Computation has no notion of instruction order, the token mechanism was required to supply the ordering information. Note that the static ordering implied by the tokens does not preclude the use of dynamic disambiguation. In fact, our simulated implementations use a load-store queue (LSQ) [SS95] very similar to the one used by a superscalar processor, described in Section 6.5.3.

Currently all data resides in a monolithic memory. This choice, although detrimental for performance, tremendously simplifies the program implementation. Since in C the address of any object can be stored and passed almost anywhere, the compiler has to maintain roughly the same memory layout as on a conventional machine for all objects whose "scope" cannot be restricted precisely statically.

Using the call-graph slightly optimizes memory layout by transforming both `static` local variables and local variables of non-recursive functions whose address is not taken into global variables. The latter transformation is useful because, since the local is no longer stack-allocated, its address becomes a link-time constant.

Breaking the global memory into (disjoint) pieces and co-locating some of these pieces closer to the computation accessing them, and the implementation of a coherence mechanism, are important subjects of future research.

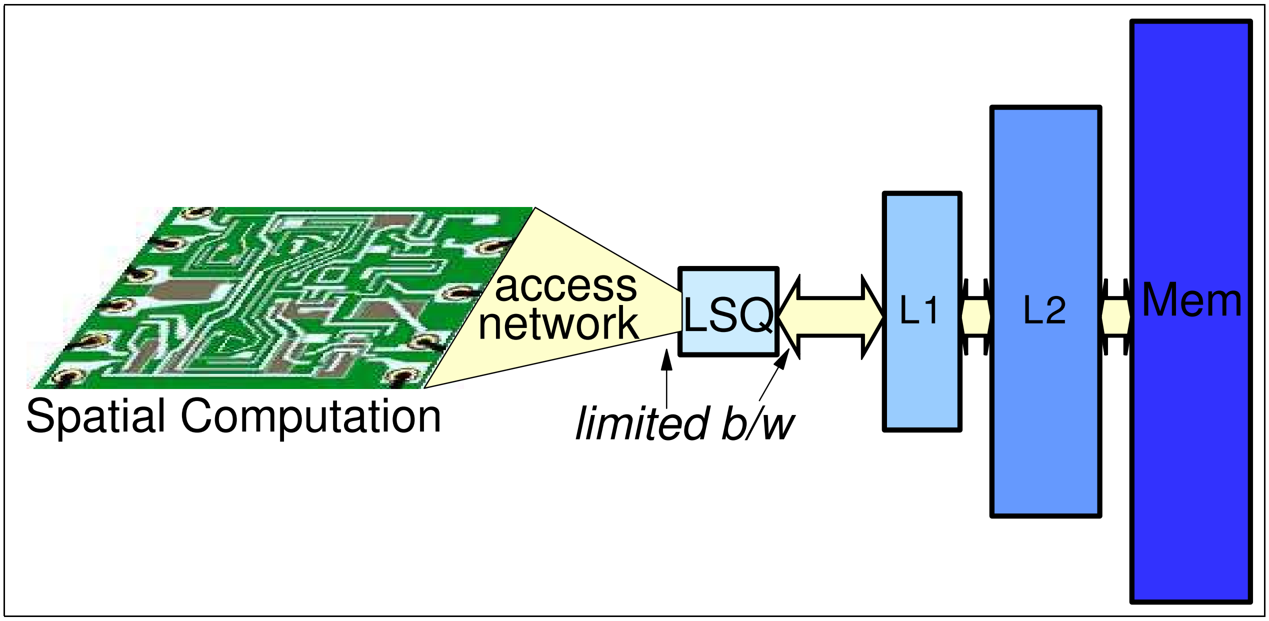


Figure 6.7: *Memory access subsystem abstract architecture.*

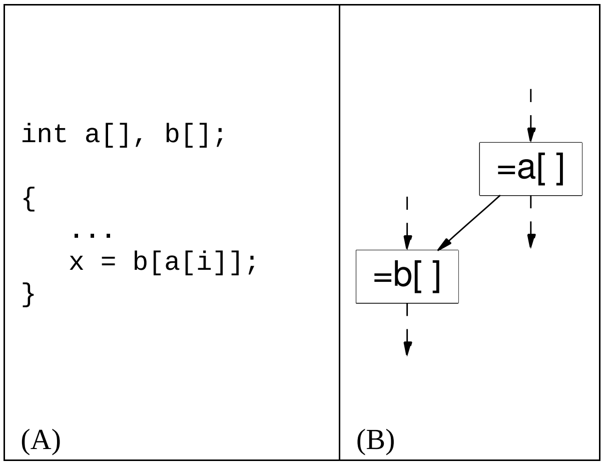
6.5.1 Architecture

Figure 6.7 shows the architecture of the memory-access subsystem employed in the circuits generated by CASH. Currently all memory operations send their requests and receive their results from a single load-store queue. For the future we are contemplating a distributed implementation for the LSQ (e.g., one LSQ for each procedure). The LSQ has two bandwidth-limited connections: one to the circuits, used to send/receive requests from load and store operations, and one connecting to the cache. When there is insufficient bandwidth to accommodate a series of simultaneous requests, an arbitration mechanism is used to grant some accesses while the rest are stalled.

6.5.2 Memory Access Protocol

As in superscalar processors, we have separated the memory access into two distinct actions: issuing the memory operations in the dependence-mandated order, and actually performing the memory access. On receipt of a token a memory operation immediately sends a *reservation request* to the LSQ. If the LSQ is not full, it reserves a slot for the memory request. This slot is filled with whatever information is available about the access itself: address, data, size of data, load/store operation identity (i.e., where to return the result), predicate. Some of the fields may be unknown at reservation time. When the slot reservation succeeds, the operation issues its token output, therefore enabling its successors to reserve slots themselves. Given enough bandwidth to the LSQ and enough storage space in the LSQ itself, the token part of the computation will propagate ahead of the scalar computation. The token's progress is slowed down by the token-propagating eta operations which require predicates in order to forward their data. Procedure calls may also cause delays in the token propagation, due to their non-lenient nature (see also Section 8.2.2.4).

When a memory operation receives more inputs, it sends updates to the LSQ to fill the incomplete slots in the reservation request. In general a memory operation requires multiple messages to the LSQ in order to complete a single access. This protocol is correct even if messages are batched. At one extreme, a single



```
int a[], b[];

{
   ...
   x = b[a[i]];
}
```

Figure 6.8: *Code fragment which can lead to deadlock involving the LSQ.*

message can be sent when all data is available. However, in general batching increases the critical path and is detrimental to performance. Therefore there is a trade-off between LSQ bandwidth usage (by sending update requests) and performance. Operations which already have an entry in the LSQ should be given priority by the memory network arbitration logic when sending updates, — otherwise deadlock can occur. An operation may reserve an LSQ slot only to discover later that its predicate is false. In this case, it sends a message to the LSQ to release its reserved slot.

Unless care is taken, this memory access protocol may lead to deadlock due to the finite capacity of the LSQ. Consider the example in Figure 6.8. Let us assume that the LSQ has only one slot. Let us also assume that the `b[]` load operation gets a token before the `a[]` load. The `b[]` load proceeds to reserve the only LSQ slot and next waits to gets the actual address. The `a[]` load operation cannot get a slot since the LSQ is full, and therefore the `b[]` load request cannot update its slot with the address information.

In order to avoid deadlock as a post-processing phase, the compiler inserts an additional token edge `a[]->b[]`. This edge does not constrain the parallelism since there is already a true dependence. However, it enforces the correct ordering between the loads from `a[]` and `b[]`. This token closure step is computed using a dataflow analysis: if there is a scalar dependence between two memory operations, but no token path, a token edge is inserted.

6.5.3 Load-Store Queue Operation

The load-store queue is used both as a small cache with high associativity and for performing dynamic memory disambiguation. It shares many similarities with the same structures used in superscalar processors. In the case of Spatial Computation an operation is speculative if the predicate controlling it is unknown. Such operations may be issued to the LSQ. Speculative loads may commit if they reach the head of the LSQ. But exceptions generated by a speculative load waits for the resolution of the speculative state. Stores cannot be committed unless their predicate is known to be true. (However, even a speculative store can be overwritten by a subsequent non-speculative store.)

There is a third type of request for the LSQ besides loads and stores: *flushes*. A frame pop operation inserts a “flush” request in the LSQ. A “flush” prevents reordering of requests inserted before and after it. When a “flush” reaches the end of the LSQ it signals to the pop operation that it can safely complete. This protocol is necessary because the “pop” operation cannot be considered complete unless all loads and stores operating from the popped frame have completed themselves.

In a hybrid system featuring a processor and a Spatial Computation fabric, control transfers from one entity to the other also need to make use of “flush” requests to establish coherence. For example, when ASH calls a procedure on the processor, it first needs to flush its own LSQ so that when the processor receives control it has a coherent view of memory. This is also true of “return” instructions which transfer control back to the processor.

Despite similarities in their functioning, there is a big difference between the way ASH and a superscalar processor access memory. In a superscalar processor the LSQ is monolithically integrated within the processor core. Control signals can be easily shared between the two. There is also a fixed number of load-store functional units, which are directly connected to the LSQ. The bandwidth of the LSQ is therefore limited. But there is no need for arbitration because the number of instructions that can issue simultaneously is fixed. When the LSQ is full, it stalls the load-store unit.

In contrast, our model of Spatial Computation “unrolls” the computational structure but keeps memory monolithic. There may now be an unbounded number of operations simultaneously trying to access memory, which is also relatively far. The interconnection medium between ASH and memory must be a relatively complex network, which requires arbitration between the concurrent accesses and dynamically routes the replies from the LSQ. While in a processor a write request can complete in effectively zero cycles, by forwarding the stored data to all subsequent pending loads of the same address, a distributed implementation requires a non-trivial amount of overhead for communication alone.

A second more subtle problem introduced by the distributed nature of the LSQ and its access network is the creation of races in memory access. For example, a read operation may send an update for its LSQ entry with a “false” predicate, indicating that it does not need a result. But in the meantime the access has completed speculatively and the result is already on its way back. The actual hardware implementation for correctly handling all corner cases of the protocol is likely to be non-trivial.

The current memory access scheme can be improved in several respects:

- First, the LSQ could be made aware of the sparse dependences between accesses. Right now, the LSQ we use still tries to enforce operation completion in insertion order: i.e., if the addresses are not yet known, there is no reordering. The LSQ is usually a fully associative structure and is therefore limited in size. By passing static dependence information to the LSQ, we envision a possible banked implementation in which conflicts are checked only within the same bank — leading to higher capacities and bandwidths for the LSQ. We also plan to study the applicability of a banked LSQ in traditional superscalar processors.

- Second, the compiler ought to partition the memory space and allocate some data to local memories, connected through high-bandwidth, low latency links. For example, local variables whose address is taken in a non-recursive procedure, but which do not escape the procedure can be allocated to a separate fixed-size memory space instead of the traditional stack frame. More sophisticated pointer and escape analysis, or even user annotations, may be used to find other candidate structures for local storage.

6.6 Dynamic Critical Paths

In Chapter 7 and Chapter 8 we compare the performance of Spatial Computation with superscalar processors on embedded and control-intensive benchmarks respectively. Understanding performance problems in complex systems is a difficult endeavor. For instance, should they be ascribed to the compiler or on the execution engine? In order to answer such a question, we need good analysis tools. The best characterization of performance of a program's execution is given by the dynamic critical path. The dynamic critical path is simultaneously a function of program dependences, execution path (which is a function of the data) and hardware resources. Despite its apparent simplicity, only recently has a clear methodology been proposed to extract and analyze dynamic critical paths in superscalar processors [FRB01, FBD02].

The problem of tracking the critical path in Spatial Computation is somewhat simpler than for superscalar processors, since there are virtually no resource constraints (except the LSQ bandwidth). We have developed two methods for discovering and visualizing the dynamic critical path, both based on [FRB01].

- The first method is based on capturing and post-processing a complete execution trace of a single selected procedure. A simulator is instrumented to dump all relevant events affecting operations in that procedure. These are sent through a pipe to a series of Perl scripts. The first Perl script filters the trace, keeping for each operation only the *last arrival event*. This is the last event which enabled the computation to proceed. Most often, this is the last input to reach an operation. However, for lenient operations the last arrival enabling computation may not coincide with the last input. For circuits experiencing backlog the last arrival event may be the acknowledgment signal which enables the computation to proceed. After computing the last arrival the trace is reversed and a continuous chain of last arrival events is computed starting with the `return` operation. This chain is the dynamic critical path. Since the chain may include looping, it is summarized as a list of static edges — each with a execution count indicating how many times that edge was on the dynamic critical path. Finally, another Perl script can combine this information with a `dot` drawing, coloring each edge with a color proportional to its frequency (the frequency is the number of critical executions of an edge divided by the total number of critical edges). An optional post-processing step can be used to display only the critical edges above a given threshold.

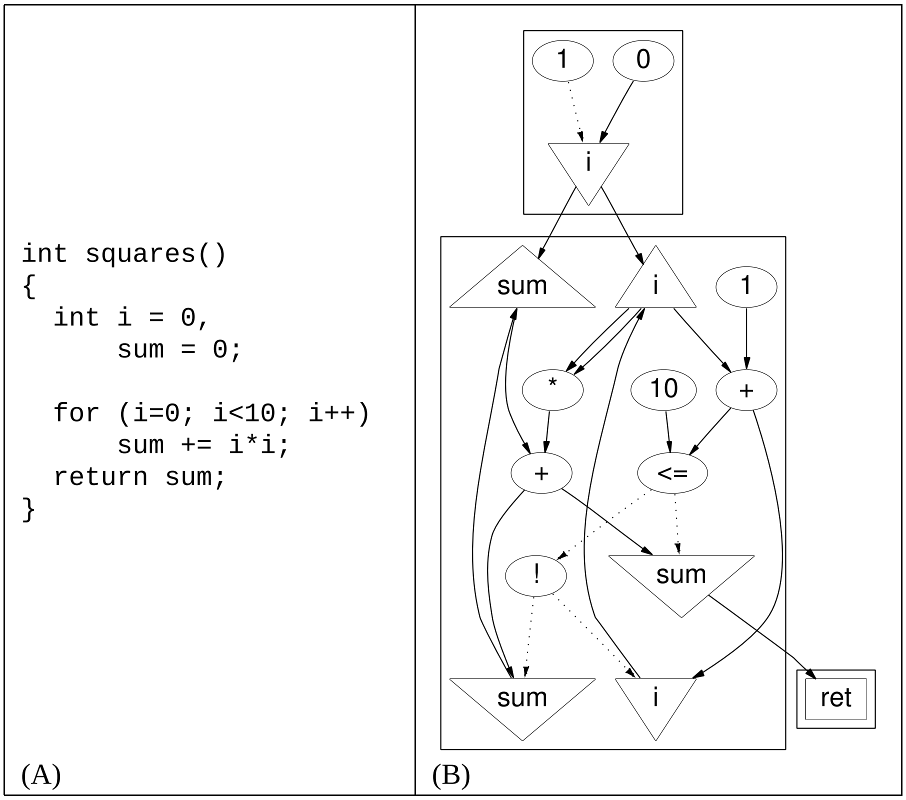
- A second method uses the on-line token propagation method from [FRB01]. Our current implementation is exact, i.e., does not use sampling. Compiler-selected operations are tagged to inject criticality tokens into the computation. A token vanishes if it travels an edge which is not last arriving. Tokens are propagated with all events originating from a critical operation. A token that survives to the "end" of the computation must have originated at a critical operation.

Figure 6.16 and Figure 6.18 below show sample critical paths as highlighted using these tools.

6.7 Dataflow Software Pipelining

In this section we analyze one of the most interesting properties of Spatial Computation — its automatic exploitation of pipeline parallelism. This phenomenon is a consequence of the dataflow nature of ASH and has been studied extensively in the dataflow literature. In the case of static dataflow machines (i.e., having at most one data item on each edge), this phenomenon is called dataflow software pipelining in [Ron82, Gao86], and in the case of dynamic dataflow loop unraveling [CA88]. The name suggests the close relationship between this form of pipelining and the traditional software pipelining optimization [AJLA95], customarily used in VLIW processors to speed-up scientific code.

We use the simple program in Figure 6.9 to illustrate this phenomenon. This program uses `i` as a basic induction variable to iterate from 1 to 10, and `sum` to accumulate the sum of the squares of `i`. Let us



```
int squares()
{
  int i = 0,
      sum = 0;

  for (i=0; i<10; i++)
      sum += i*i;
  return sum;
}
```

Figure 6.9: *Program exhibiting dataflow software pipelining. We have omitted the token, crt and pc networks.*

further assume that the multiplier implementation is pipelined with eight stages. In Figure 6.10 we show the snapshots for a few consecutive "clock ticks" of this circuit.

Notice that the original CFG loop has been decomposed into two independent dataflow loops, shown in Figure 6.11: one computing the value of `i` and the other computing the value of `sum`. The two loops are connected by the multiplier, which gets its input from one loop and sends its output to the second loop. In the last snapshot in Figure 6.10, the `i` loop is more than one iteration ahead of the `sum` loop.

A similar effect can be achieved in a statically scheduled computation by explicitly software pipelining the loop, scheduling the computation of `i` to occur one iteration ahead of `sum`.

However, let us notice some important differences between the two types of software pipelining:

- Traditional software pipelining relies on precise scheduling of the operations for each clock cycle. Unpredictable events and unknown latency operations (such as cache misses) do not fit well within the framework. In contrast, Spatial Computation requires no scheduling and automatically adjusts when the schedule is disrupted because the execution of all operations is completely decoupled.

- Effective software pipelining relies on accurate resource usage modeling by each instruction, some of which are very hard to reason about, such as cache bank conflicts [RGLS96]. In Spatial Computation as many resources as necessary can be synthesized. Unpredictable contention for the shared resources, such as LSQ ports or cache banks, has only local effects on the schedule.

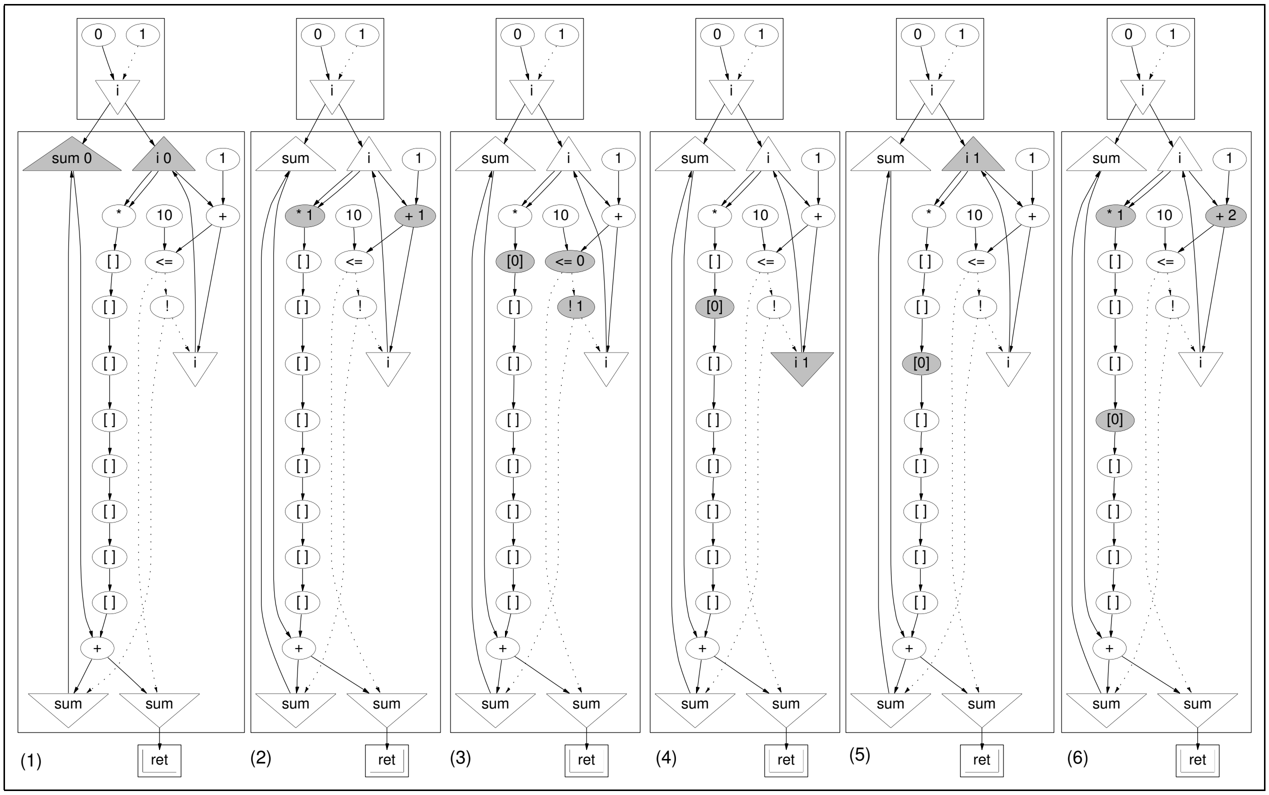


Figure 6.10: *Snapshots of the execution of the circuit in Figure 6.9, assuming that the multiplier implementation is pipelined with eight stages and that the negation latency is zero. In the last snapshot, two different values of* `i` *are simultaneously present in the multiplier pipeline.*

- According to [RGLS96], sophisticated production-quality implementations of software pipelining require a huge amount of code. The MIPS R8000 implementation described in that work has 33,000 lines of code, which is more than the complete CASH core.

6.7.1 Pipelining Opportunities

All studies of software pipelining that we are aware of have studied its applicability in the context of scientific computation. We have focused our attention on analyzing the opportunities of pipelining loops in C benchmarks. We have compiled C programs from our benchmark suites and analyzed the structure of the innermost loops. It is easy to statically detect the occurrence of dataflow software pipelining, at least for innermost loops. Whenever the Pegasus representation of an innermost loop contains more than one strongly connected component (SCC),[3] the resulting components are disjoint loops which can advance at different rates. Computing SCCs is a linear-time operation using Tarjan's algorithm [Tar72]. Figure 6.11 highlights the two SCCs for the loop of the program in Figure 6.9.

In Figures 6.12—6.14 we show for each benchmark a distribution of the number of non-trivial SCCs per loop, found in the hot functions. We have only analyzed the functions where the program spends at least 1% of its running time. Tarjan's algorithm labels each node not part of a cycle as a "trivial" SCC.

[3] A strongly connected component of a directed graph is a set of nodes where a directed path between any two nodes in the set exists.

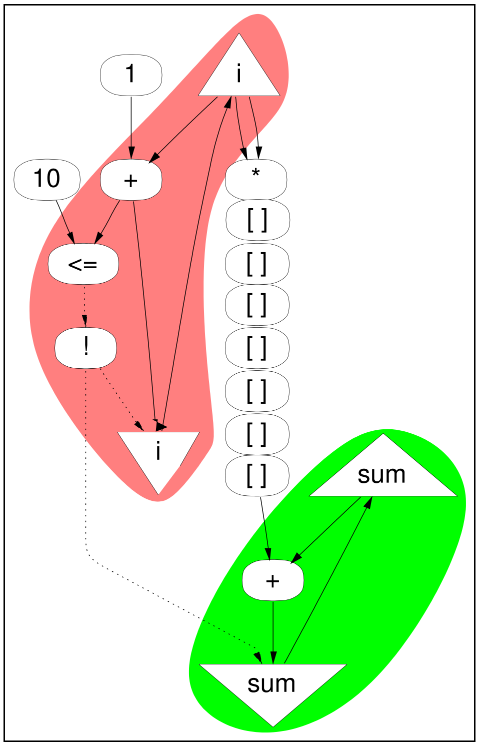


Figure 6.11: *Strongly connected components of the circuit in Figure 6.9.*

However, note that in all Pegasus graphs the loop-invariant values give rise to “empty” SCCs, which we also catalog as “trivial.” For example, all loops contain an “empty” SCC for carrying around the program counter operation. As expected, in almost all programs the number of non-pipelineable loops (i.e., with only one SCC) dominates. It is impossible to translate these static numbers directly into performance. What matters is, of course, how pipelineable the “hot” loops are, as well as the *initiation interval*.

An interesting fact is that there are loops with a very large number of non-trivial SCCs. The maximum is 18, in 186.crafty. A large number of SCCs does not translate necessarily into increased performance since (as described below) the average computation rate (i.e., the equivalent of the reciprocal of the initiation interval) is given by the “slowest” of the SCCs. The most uniform distribution of SCCs is shown by Mediabench. The number of SCCs is a rough measure of the data parallelism of a program. We indeed expect the programs from Mediabench to exhibit a greater amount of data parallelism than the other benchmark suites.

By collapsing each SCC into a single node one obtains a directed acyclic graph. In a topological sort of this graph, the “first” SCC is always the one computing the loop predicate. This predicate is used by all other SCCs to decide completion. If the number of SCCs increases when ignoring the token edges, memory dependences are the inhibiting factor for pipelining.

6.7.2 Pipeline Balancing

Dataflow software pipelining is desirable because it increases the throughput of the computation without any additional hardware costs. However, the pipelining that occurs naturally is often inhibited by the graph structure. Let us take another look at the computation in Figure 6.10. Figure 6.15 shows the state of that computation for the next two clock ticks.

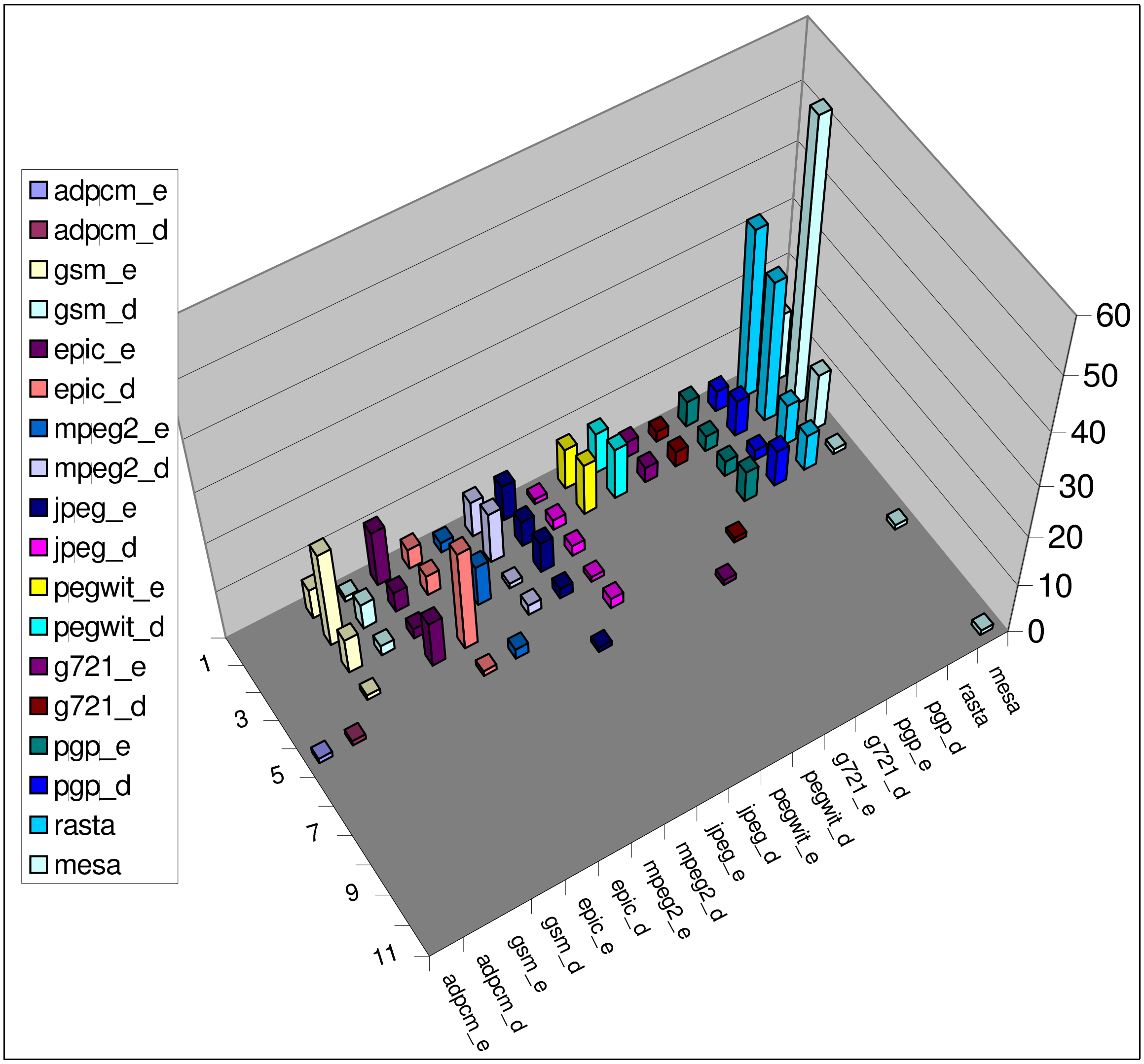


Figure 6.12: *Histogram of the number of strongly-connected components per innermost loop in the hot functions of the Mediabench benchmarks.*

While data continues to flow through the multiplier, the data in the `i` loop is blocked. The reason for the lack of progress is the fact that both the comparison `<=` and the negation `!` have not received acknowledgments from some of their outputs. Neither of the `sum` eta nodes has sent an ack because neither has yet received the other input — which is expected from the multiplier output. The `i` loop will be stuck until cycle 11, when the output of the multiplier goes to both eta nodes and the acknowledgment is finally received.

Another way to look at what happens in this circuit is to analyze the dynamic critical path. We have shown it schematically in Figure 6.16. In the steady state, the critical path contains three acknowledgment edges: (`sum` eta)←(`!`)←(`<=`)←(`+`). The last event to reach the negation is not its input, but the acknowledgment. This state of affairs is rather unfortunate since our intuition tells us that this circuit ought to benefit fully from pipelining. But in order to do so we need to perform a transformation dubbed “pipeline balancing” by Guang Gao [Gao86, GP89, Gao90]. A very related transformation in the realm of asynchronous

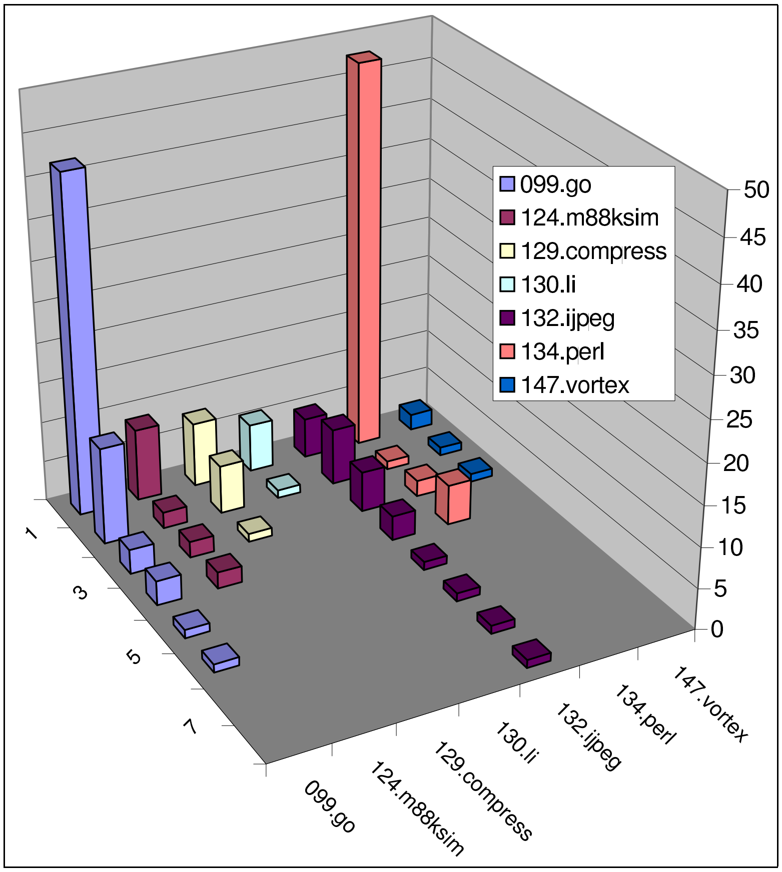


Figure 6.13: *Histogram of the number of strongly-connected components per innermost loop in the hot functions of the SpecInt 95 benchmarks.*

circuits is dubbed “slack matching” [Lin95].

Gao considers only the case of a static dataflow DAG with a single source s and sink t, where the source can produce data continuously and the sink can consume data immediately. In his model all nodes have the same propagation delay. Lines’ work adopts a much lower-level model and describes how matching is to be performed for certain circuit topologies, but does not treat the general case.

Consider two paths from the source joined at some internal node n, as in Figure 6.17. If there are a different number of *latches* on these two paths,[4] there is an imbalance: after data traveling the shorter path reaches n, s is blocked because n will emit an ack only after the data on the long path reaches it. s is backlogged by the lack of acknowledgments. If the lengths of the two paths are l_1 and l_2, the source can inject l_1 new requests only every l_2 time units and the computation rate is l_1/l_2. By adding $l_2 - l_1$ delay elements on the short path the delayed ack problem goes away: data generated by the source is stored within the delay elements and these can send the acknowledgment immediately. Gao’s algorithm makes all paths within the DAG of the same length achieving maximal throughput. He shows that optimizing for a minimal number of delay elements can be achieved by solving a linear program.

The inserted latches are the direct counterpart of the *reservation stations* [Tom67] used in superscalar processors to queue ready operations for a functional unit.

[4] Remember that each operation output is latched.

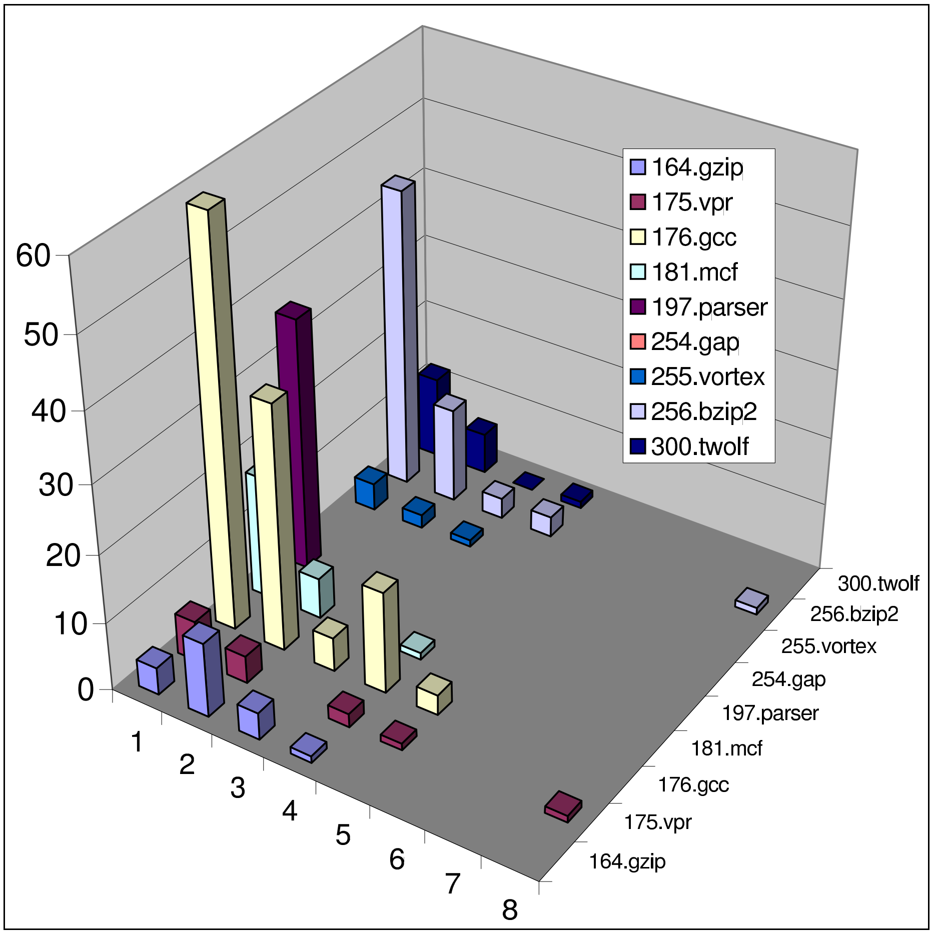


Figure 6.14: *Histogram of the number of strongly-connected components per innermost loop in the hot functions of the SpecInt 2000 C benchmarks.*

However, the problem we are facing is substantially more complicated:

- Operations have varying computation delays. As we show below, this is an important difference which cannot be handled by the simple model.
- Our computation graphs contain cycles.
- Lenient nodes may generate outputs before all inputs have been received.

While we believe that a precise analytic solution for coping with the first two limitations can be found, the last one is arguably much harder to handle because lenient operations have data-dependent latencies.

Note that we are optimizing here for throughput and not for latency. If the loop has a small number of iterations, it may be that balancing worsens its performance. We are only considering the asymptotic behavior of the loop when it executes a very large number of iterations and it reaches a steady state. Under the assumption of fixed latency operations, the behavior of such a loop becomes periodic after awhile [GWN91], although determining the actual period is difficult in the most general case. We can talk about the “average period” of a computation, which is the period in which any node will produce a new output.

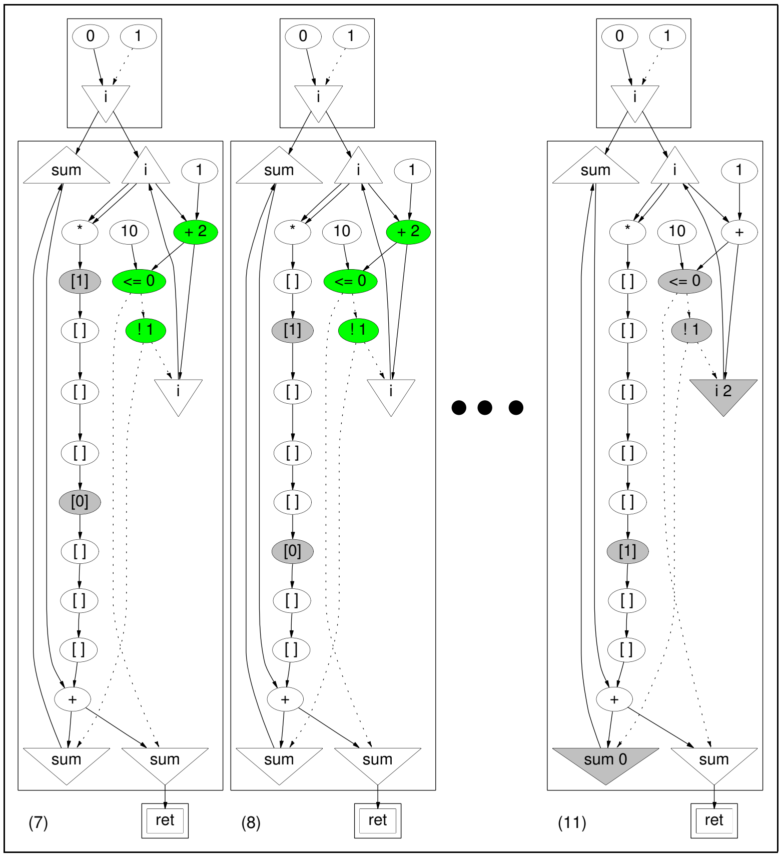


Figure 6.15: *Evolution of the circuit in Figure 6.10.*

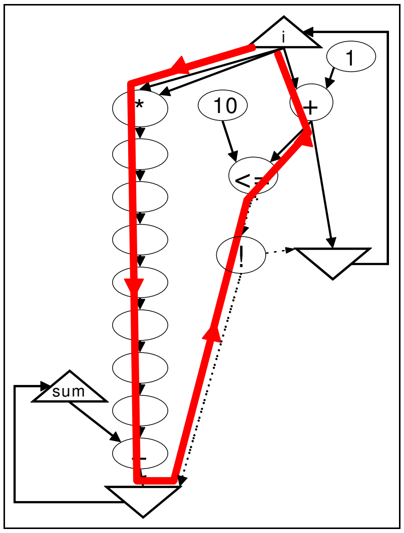


Figure 6.16: *Dynamic critical path in the loop steady state for the circuit in Figure 6.9.*

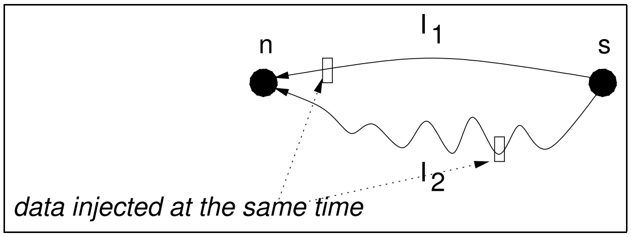


Figure 6.17: *A scenario requiring pipeline balancing.*

6.7.2.1 The Initiation Interval

First, let us analyze the impact of cycles in the computation graph. Each SCC contains at least one simple cycle. Each simple cycle represents a true loop-carried dependency, which therefore gives a lower bound on the computation rate. The period of an SCC is directly related to the length of the longest cycle. However, it is not always the longest cycle because an SCC may contain multiple "mu" nodes and therefore generate multiple data values in one complete loop iteration.

In a circuit containing multiple SCCs (such as the one in Figure 6.9) — while the different SCCs may have different periods — asymptotically all SCCs will end up computing at the same rate, the smallest of all the rates (i.e., longest period). This occurs because the SCCs "downstream" of the slowest one receive data at this rate and the SCCs "upstream" receive acknowledgments at this rate. This computation rate corresponds directly to the initiation interval notion of classical software pipelining. Unlike the classical initiation interval, which is a compile-time constant, the computation rate is a dynamically varying value depending on the multiplexor control signals.

6.7.2.2 Long Latency Operations

Let us analyze the impact of long-latency operations. Since an operation cannot acknowledge some inputs before having computed and stored the corresponding output, it follows that the period of the whole computation is *bounded below by the longest latency* operation. This latency can be reduced by pipelining and/or predicating long latency operations. For example, division is predicated by CASH. If a division should not be executed at run-time, (and therefore should not be on the dynamic critical path), its predicate is "false" and its run-time latency is effectively zero[5] because the implementation is lenient in the predicate input. However, pipelining a long-latency operation which is part of an SCC does *not* increase throughput since it does not decrease the SCC latency. Pipelining an operation is also wasteful if the operation latency is already smaller than the period of the slowest SCC.

6.7.2.3 Pipeline Balancing Revisited

The generalized pipeline balancing algorithm used by CASH is the following:

- The Pegasus graph is decomposed into SCCs.
- For each SCC s the average period P_s is computed ($P_s =$longest simple cycle/number of mu operations in cycle). $P = \max_s P_s$ is the longest average period.
- The latency of each operation is L_o; $L = \max_o L_o$ is the longest latency (non-pipelined) operation.

[5] Counting at the moment the predicate input is received.

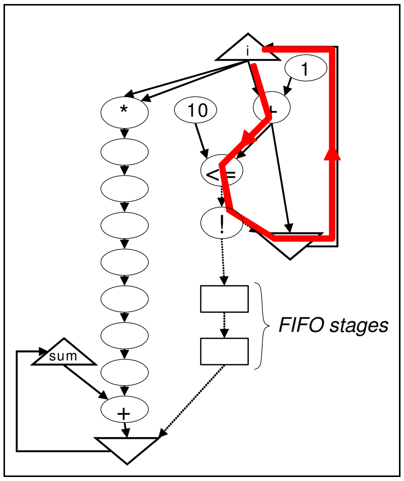


Figure 6.18: *Dynamic critical path in the loop steady state after pipeline balancing for the circuit in Figure 6.9. The square operations are FIFO buffers.*

- The computation period of the whole hyperblock is $T = \max(L, P)$.
- All SCCs are collapsed to single nodes, creating a DAG.
- The DAG is balanced by inserting *elastic pipelines* [Sut89]. These are cut-through registers, i.e., which immediately forward data if the output channel is free or otherwise latch it. Elastic pipelines are modeled in Pegasus as simple "no-ops."
- The pipelining algorithm estimates the arrival time of the inputs to an operation, say t_1 and t_2. (SCCs are considered operations with multiple inputs and outputs.) $\Delta = |t_1 - t_2|$. if $\Delta > P$, $\lceil \Delta/P \rceil$ elastic pipeline stages are inserted on the earlier input.

For the example in Figure 6.9, the `i` SCC has a period of four, while the `sum` SCC has a period of three (assuming the mu and eta take one cycle each). If the multiplier is not pipelined the period is eight. Pipelining the multiplier with stages one-cycle deep is somewhat wasteful, as seen in Figure 6.10. Since the rate of the `i` loop is four, there are three bubbles between two data items in the multiplier.

In the collapsed graph there are two edges connecting the `i` and `sum` SCCs: one with latency eight (the multiplier), and one with latency zero (the predicate). At a period of four, two FIFO stages are enough to balance the predicate path. Figure 6.18 shows the circuit built by CASH after pipeline balancing, with the dynamic critical path highlighted. The critical path in the steady state follows exactly the longest cycle. This circuit cannot compute any faster.[6] In this circuit the `i` loop is completely independent and can be ahead of the `sum` loop with two iterations. The two predicates are computed ahead and stored in the FIFO buffers. There are no more acknowledgment edges on the critical path.

In general, one cannot remove all ack edges from the critical path. If the `sum` loop would contain a longer-latency computation, it would be the critical loop, forcing the `i` loop to slow down through the release of infrequent acknowledgments.

We can now rephrase all the memory optimizations described in Section 4.5: their purpose is to divide some of the SCCs into multiple SCCs, increasing the opportunities for parallelism. Loop decoupling is the

[6] Unless the dataflow limit is circumvented by prediction (see Chapter 8).

equivalent of the traditional software pipelining operation of skewing the computation by an amount less than the dependence distance.

6.7.2.4 Resource Sharing Revisited

Having observed the run-time behavior of ASH circuits, we can revisit some of the core implementation decisions. Our model of Spatial Computation was designed to completely avoid hardware resource sharing (on a per-procedure basis) for two reasons: (1) not to hinder parallelism by requiring dynamic resource scheduling; and (2) to allow direct, non-arbitered communication between data producers and consumers.

Consider a strongly connected component with a single “mu” or “switch” node, i.e., a “slice” of a loop representing a true loop-carried dependence. At any point in time there is exactly one live data item in this SCC.[7] Therefore, there is no parallel execution within the SCC and a hardware implementation sharing resources may be advantageous. The downside is a more complex interconnection network, which must support switching. Depending on the actual cost of the computational units, it may or may not be profitable to share resources.

6.8 Unrolling Effects

Unless we otherwise specify, all results presented make use of very aggressive loop unrolling. We unroll both `for` and `while` loops, having both known and unknown loop bounds. In most cases loop unrolling increases performance of the ASH implementation because it increases the ILP within each iteration and often it can amortize some of the control-flow computation. Notice that unrolling occurs in the front-end (see Figure 2.4) before the Pegasus representation is even built, and therefore uses very little information about the code structure. We have inhibited the unrolling of loops containing procedure calls and expensive non-pipelined operations, such as division. Unrolling loops with unknown bounds and conditional expensive operations can create large hyperblocks with many copies of a long-latency operation. Since all operations in a hyperblock have to execute to completion, the latency of the unrolled loop body hyperblock is very large. If the long-latency operation is only conditionally executed, the overhead of the execution of the unrolled body may be very large.

[7] Except possibly during the last iteration, when a value corresponding to the *next* outer iteration may enter before the last one has left.

Chapter 7

ASH in Embedded Systems

Chapters 7 and 8 are devoted to performance evaluations of Spatial Computation. This chapter uses CASH in the domain of media processing for embedded systems. The goal is to evaluate the benefits of this model of computation for important media kernels ("important" in the sense of requiring a majority of the computation cycles). The next chapter will be devoted to the evaluation of ASH for more generic computational tasks.

7.1 Introduction

First, let us clarify our premises about the domain of embedded computation. We assume that we are given a standard algorithm for some form of data-intensive processing, e.g., movie decoding, image processing or speech encoding. The goal is to implement the algorithm using minimal resources and highest performance. The application is assumed to be fairly stable, warranting a custom hardware or reconfigurable hardware implementation. The application is specified with a reference C implementation and has some compute-intensive kernels.[1] We also presume that the kernel selection is made manually and at the granularity of whole functions (see Section 7.2). Since the application set is relatively narrow, we envision a development loop where the system designer can tweak the compilation options in the attempt of extracting maximum performance. Moreover, since the lack of precise pointer information is one of the main performance inhibitors, in Section 7.3 we allow the programmer to manually annotate the application code with `#pragma` statements to convey inter-procedural pointer information to the compiler. We do not perform any other code restructuring.

We target a system-on-a-chip type of architecture, consisting of a typical low-power embedded microprocessor tightly coupled with specialized support hardware, as illustrated in Figure 7.1. Both access the same memory hierarchy and have a coherent view of it. The support hardware structure is fairly small and is used to accelerate the important kernels. We call the supporting hardware structure a Hardware Accelerator (HA) structure, be it a co-processor, a reconfigurable hardware fabric or custom hardware circuitry. CASH is used in this context as a quick prototyping tool for generating the HA.

The main arguments supporting the use of such a tool-chain are: (1) very fast time-to-market due to very fast design exploration and circuit synthesis; (2) low power because of the asynchronous nature of the support circuitry. (The power consumption is expected to be directly proportional with the amount of

[1] Note that the benchmarks we use have relatively small input sets. E.g., the `mpeg2` encoder we use only processes three frames. For larger input sets we expect the hot spots of the program to become even more prominent.

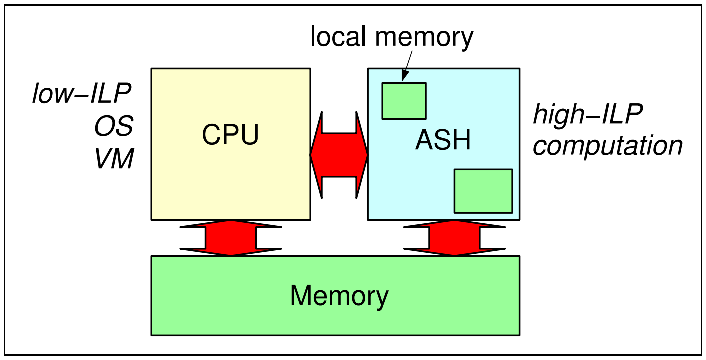


Figure 7.1: *System-on-a-chip architecture for using Application-Specific Hardware.*

useful computation. High parallelism and speculation may drive power consumption up.) and (3) high performance due to the lean nature of the synthesized circuits.

Let us examine the circuits generated by CASH. The only global structure of the circuits is the memory access interface and the network connecting it to the memory operations. There is no register file, monolithic or clustered. All latches have a single write and a single read port. All data connections are point-to-point and have a single writer. We expect the synthesized circuits to have good layouts, requiring mostly the use of very short wires.[2] This last aspect is particularly important, since in future technologies the delay and power consumption on medium and long wires dominate the logic devices (see Figure 1.2).

Moreover, there is *no global control circuitry*: control is completely distributed within the computation, in the ready/acknowledge signals, and through the predicates driving muxes and etas. There are no associative memories, no complex bypass networks, no prediction, no instructions and none of the associated structures: instruction cache, dispatch, decoding, commit logic. Therefore, it is fair to characterize the resulting circuits as being quite "lean." As a result, we expect that ASH can be synthesized to run at a fairly high speed. In the evaluation that follows we assume that ASH and the processor have comparable absolute latencies for all arithmetic and logic.

7.2 Hardware-Software Partitioning

An important question in coding applications for a hybrid system-on-a-chip is how the application is partitioned into hardware and software components. Our past work [BMBG02] has suggested an approach to this difficult problem, which we think considerably simplifies it. We will review it here briefly, since it is the basis of our evaluation.

We argue that HA should be integrated in a computing system not as a subordinate of the processor, but as an equal peer. Moreover, we propose a *procedural interface* between software on the processor and the HA in the style of Remote Procedure Calls [Nel81]. Under our proposal processor-resident programs can invoke code on the HA in the same way they can invoke library functions. HA-based code should also be able to call code on the processor. Our proposal is not a panacea for solving the problem of hardware-software partitioning, since we are proposing a *mechanism* and not a *policy* for how the two sides of an application should be interfaced. However, we believe that the choice of a good interface is extremely important for unleashing the full potential of a new computing paradigm: witness the success of interfaces such as libraries, system calls, remote procedure calls and sockets.

[2] However, the CAD tools may chose bad layouts, which will stretch some signals.

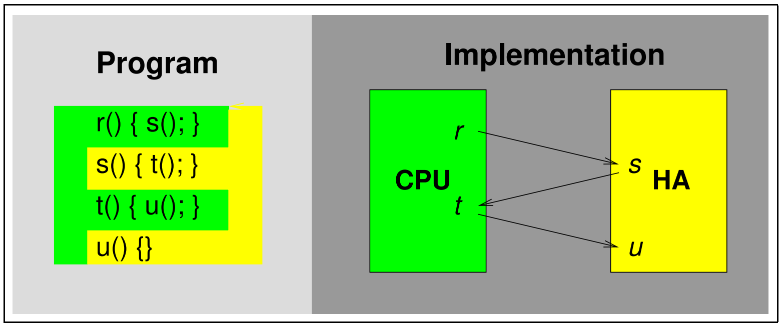


Figure 7.2: *A peer-to-peer hardware-software system enables procedure calls to proceed in either direction.*

In [BMBG02] we propose to use a hardware-independent and language-independent hardware-software interface similar to remote procedure calls, which can be used between the code executed on a processor and the code executed on a HA. We also show that a compiler can automatically generate the stubs for interfacing the CPU and the HA.

We call the resulting hardware-software interface a *peer-to-peer* interface because the HA is an equal peer in the process of computation, instead of being a slave to the processor. The HA must be able to invoke processor services through procedure calls as well, as illustrated in Figure 7.2. This last requirement is a key of our proposal because it gives tremendous freedom in the partitioning of the computation into hardware and software components.

This approach is motivated by the observation that most well-engineered code may contain error handling code even in the compute-intensive kernels. Such error code is practically never executed at run-time, making these functions dynamically leaves of the call-graph, even if in the static call-graph they are not leaves (since they call error-handling code, e.g., `printf` statements). In a system where the HA cannot execute the error-handling code (for example because it requires I/O), we have to either (1) remove the error handling from the selected kernels; (2) give up on implementing the kernels altogether; or (3) enable the HA to invoke the error-handling code without being able to execute it. This last option is enabled by our proposed architecture.

Once we have agreed on a peer-to-peer architecture, the most natural way to partition a software program is exactly at *procedure boundaries*. In other words, procedures should be mapped entirely either on the CPU or on the HA.

In the rest of this chapter we assume the use of such an architecture. We use profiling information to manually select important kernels. For most applications at run-time these kernels do not make any procedure calls. We use CASH to compile the kernels to application-specific hardware and then only analyze the performance of the kernel itself. The cost and frequency of cross-service invocation will have an important bearing on the performance of the final system. However, we ignore them in this work.

7.3 Pointer Independence

One of the major factors inhibiting optimizations when compiling C is the lack of precise information about pointers and updates through side-effects. Our belief is that whole-program analysis is inherently not scalable (although we are aware of an impressive literature devoted to pointer analysis [Hin01], including fast coarse whole-program analyses [HT01]). Therefore CASH, relies only on procedure-level analyses; the pointer analysis is also intra-procedural.

We think that a language aspect which can have such an important role, not only for performance, but also for correctness,[3] ought to be made explicitly part of the language itself and given as a tool to programmers. In fact, recent dialects of C made moves in such a direction through the introduction of qualifiers such as `const` in ANSI C and `restrict` in C99. The compiler should not be responsible for inferring such annotations but could help in verifying their correctness [FAT02, AFKT03], while automated tools could be used to help the programmer find places where such annotations may be useful [AFKT03, KBVG03]. In fact, commercial compilers frequently make use of such annotations — for which a special extension mechanism has been introduced in the C language, the `#pragma` statements.

In order to allow the programmer to supply information about pointer aliasing we make use of a `#pragma independent` statement. `#pragma independent p q` guarantees to the compiler that, in the current scope, pointers `p` and `q` never point to the same object. This pragma is somewhat similar to the proposed C99 `restrict` keyword, but much more intuitive to use. A pragma is most often used to indicate that the pointer arguments to a function do not alias to each other, or that arguments and global variables do not interfere. The **Prg** column of Table 7.2 shows, for each of the kernels we are evaluating, the number of pragma statements manually inserted.

Pragma handling by the compiler is fairly straightforward. On the CFG, before building Pegasus, we use a simple dataflow analysis to propagate the independence information to all pointer expressions based on the indicated objects. All pointer expressions having `p` and `q` as "base" (i.e., derived from those by pointer arithmetic) are also deemed to be independent. This propagation is very similar to the analysis called "connection analysis" in [GH96]. Independence information is incorporated in Pegasus in a two step process: (1) the memory disambiguator (see Section 4.4.2.4) removes token edges between memory accesses deemed independent by the connection analysis; and (2) abstract read-write sets are created for the independent pointers and new "mu" and "eta" operators are created for handling their tokens.

Removal of token edges due to independence information can improve program performance due to multiple effects: (1) better register promotion, (2) more parallelism in memory accesses and (3) better dataflow software pipelining, if the removal of the token edges leads to the creation of new strongly-connected components.

7.4 Evaluation

In this section we evaluate the effectiveness of CASH on selected kernels from embedded-type benchmarks. We use programs from the Mediabench [LPMS97] benchmark suite. For selecting the kernels we first run each benchmark on a superscalar processor simulator and collect profiling information. Next we compile each kernel using CASH and the rest of the benchmark using gcc. We simulate the resulting "executable" by measuring only the kernel execution. The baseline performance is measured using the SimpleScalar 3.0 simulator [BA97], configured as a four-way out-of-order superscalar.

On the ASH system the load-store queue has more input ports, due to the high bandwidth requirements from the memory access protocol described in Section 6.5.2. The L1 and L2 data caches and the memory latency are all identical on the compared systems.

Table 7.1 shows the assumed operation latencies for the computational primitives. Most of these values are taken to match a prototypical processor, which will be the basis for our comparison. Some values are smaller for ASH, due to its specialized hardware nature. For example, we assume that constant shifts or negation operations can be made in zero cycles. The assumption for negation is justified because it is always

[3] For example, the standard C library function `memcpy` explicitly requires that there is no overlap of the source and destination argument arrays.

a one-bit operation and we assume that in a hardware implementation, the source of the negation operation always computes the correct value and also the complement. We assume that the timing of most arithmetic operations is the same on both systems, although this is probably unfair to the processor since its arithmetic units are highly optimized, trading-off area for performance.

In CASH we use very aggressive loop unrolling and we disable inlining, unless otherwise specified. We use the full complement of the optimizations of the compilers for both systems.

Note that since ASH has no instruction issue/decode/dispatch logic, it cannot be limited by instruction processing. The I-cache of the processor has a one-cycle hit time.

7.4.1 Kernels

Table 7.2 shows for each program the functions we manually selected, their size and contribution to the program run-time, and the number of `pragma independent` statements added manually to help pointer disambiguation. We have selected these kernels by profiling the programs on a real processor and selecting the functions with the highest profile count. The only exception to this rule was the function `encode_mcu_AC_refine` which is the hottest function from `jpeg_e`, but which makes an inordinate amount of function calls in the innermost loop,[4] and therefore was excluded. For both `g721` benchmarks we have manually inlined the function `quan` within `fmult`. This is the only case of a kernel containing inlined code, both in the baseline and in ASH. For both `pgp` programs we had to force unrolling to be even more aggressive and unroll the only loop in the kernel (which has unknown bounds) 16 times, instead of the default four chosen by the heuristics.

7.4.2 Performance Results

We express performance in machine cycles for executing the kernel alone. We time the kernel on the processor and on ASH. The comparison is not perfect since, due to our simulation methodology, ASH does not have the data caches warmed-up in the same way as the processor. We implicitly assume the same "clock" for both implementations. Given the low power requirements for embedded processors, this is a defensible assumption.

Figure 7.3 shows how much faster ASH is compared to the superscalar core. All kernels exhibit significant speed-ups, ranging from 50% up to 1200% for `rasta`, except one program, `epic_e` which slows down and one program `adpcm_e`, which breaks-even (with a speed-up of only 2%).

Figure 7.4 shows the indisputable advantage of Spatial Computation: unlimited parallelism, exploited post-hoc. It shows the sustained IPC[5] for both the processor and ASH for the selected kernels. On all kernels ASH sustains a higher IPC, and except for `adpcm_e`, an IPC higher than the maximum theoretically available from the processor. In fact, excepting three benchmarks (`adpcm_e` and both `g721`), the IPC of ASH is more than twice that of the processor.

IPC does not immediately translate into speed-up because of the aggressive speculation performed by ASH. In order to quantify how much of the IPC is "wasted" on the wrong path, we have plotted Figure 7.5,

[4] While most of these functions can be inlined, we could not easily control the inlining by `gcc`.

[5] Note that the traditional definition of IPC excludes the speculative instructions. The processor measure of IPC counts only non-speculative instructions, while ASH counts speculative instructions as well. However, to decide which instruction is speculative in ASH is more difficult than for a processor, since the speculative state of an instruction can be discovered a long time after the instruction has been committed. A similar problem is encountered when defining IPC for architectures featuring a "conditional move" instruction, or which employ predicate promotion or aggressive code scheduling. In these cases some "useless" instructions may nevertheless be executed on the correct execution path. An estimation of the amount of speculation in ASH can be found in Section 7.5.2.

Operation	**ASH**	**Baseline**	**Observations**
+, $-$	1	1	
FP +, $-$,	2	2	
*	3	3	pipelined at 1 cycle
muluh	3	N/A	
FP *	4	4	pipelined at 1 cycle
/, %	24	24	
FP /, %	12	12	
integer cast	0	0	
int $\leftrightarrow$ FP cast	1	1	
FP $\leftrightarrow$ FP cast	2	2	
compare	1	1	
FP compare	2	2	
long FP compare	3	3	
bitwise	1	1	
constant shift	0	1	
shift	1	1	
negation	0	1	
Boolean	$\log(in)/2$	1	Always two inputs on CPU
mux	$\log(in/4)$	N/A	
mu, switch	$\log(in)$	N/A	
eta	0	N/A	Folded into source
V	$log(in)$	N/A	
call	1	n	No stack frame in ASH
continuation	1	N/A	
return	1	1	
frame	1	N/A	Part of call handling on CPU
LSQ access latency	2	0	1 cycle each way for ASH
LSQ size	64	64	
LSQ in bandwidth	4	2	ASH needs more bandwidth
LSQ out bandwidth	2	2	
L1 cache latency	2	2	same for I-cache for CPU
L1 cache capacity	32K	32K	4-ways, LRU
L2 cache latency	8	8	unified with I-cache for CPU
L2 cache capacity	256K	256K	2-ways, LRU
I1 cache latency	N/A	1	
I1 cache capacity	N/A	32K	4-ways, LRU
I2 cache	N/A	unified	
DTLB	64	64	30 cycles latency
Memory latency	72	72	

Table 7.1: *Operation latencies for Spatial Computation.*

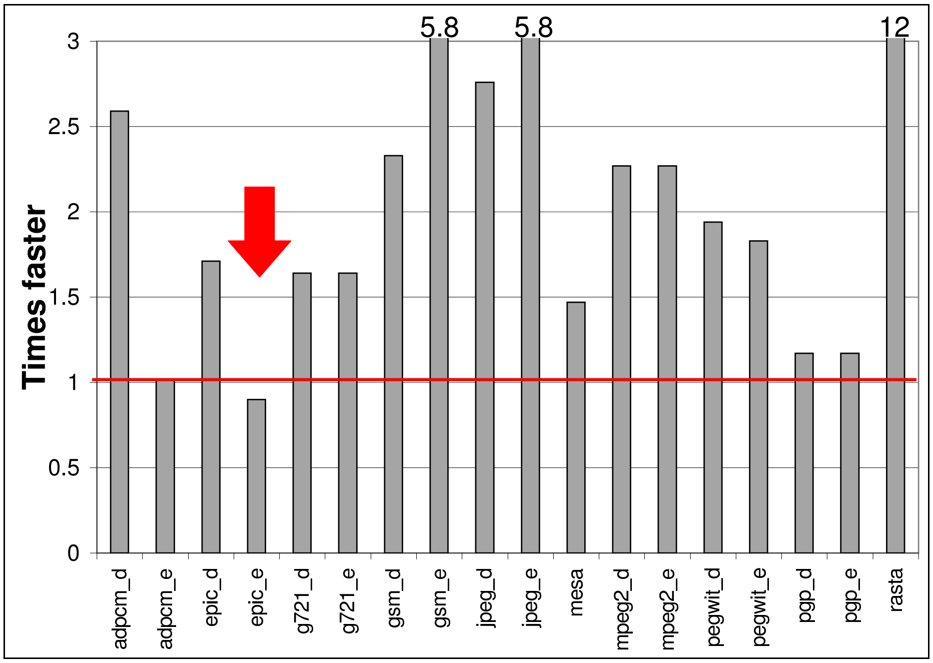


Figure 7.3: *Speed-up of ASH versus a four-way out-of-order superscalar processor. The only kernel exhibiting a slowdown is* `epic_e`.

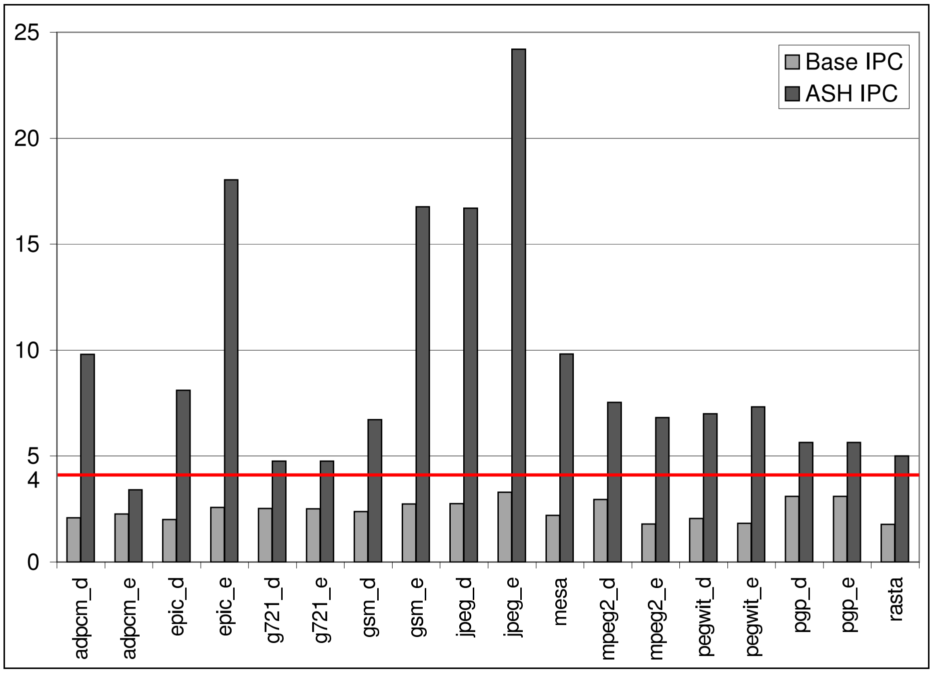


Figure 7.4: *Average IPC (Instructions Per Cycle) for superscalar and ASH. The upper limit (4) for the processor is indicated. The data for ASH includes speculatively executed instructions.*

Benchmark	Functions	LOC	Run	Prg	Observations
adpcm_d	adpcm_decoder	80	100	5	
adpcm_e	adpcm_coder	103	100	6	
epic_d	collapse_pyr	265	48	1	
epic_e	internal_filter	223	84	10	
g721_d	update	190	21	0	
	fmult+quan	22+14	30	0	quan inlined in fmult
g721_e	update	190	20	0	
	fmult+quan	22+14	30	0	quan inlined in fmult
gsm_d	Short_term_synthesis_filtering	24	71	7	
gsm_e	Calculation_of_the_LTP_parameters	137	43	6	
	Short_term_analysis_filtering	43	24	3	
jpeg_d	jpeg_idct_islow	241	49	7	
	ycc_rgb_convert	40	19	8	
jpeg_e	jpeg_fdct_islow	144	19	0	
mesa	de_casteljau_surf	287	33	6	
mpeg2_d	idctcol	54	20	1	
	Saturate	27	15	0	
mpeg2_e	dist1	92	72	1	
	fdct	29	13	0	
pegwit_d	gfAddMul	19	31	3	
	squareDecrypt	28	30	1	
	gfMultiply	41	28	4	
pegwit_e	gfAddMul	19	33	3	
	gfMultiply	41	31	4	
pgp_d	mp_smul	16	77	1	
pgp_e	mp_smul	16	63	1	
rasta	FR4TR	129	10	0	

Table 7.2: *Embedded benchmark kernels selected.* **LOC** *is the number of source lines,* **Run** *is the percentage of the running time spent in that function for some reference input,* **Prg** *is the number of `#pragma independent` statements inserted.*

which shows the obtained speed-up versus the ratio of the IPCs. Since IPC is directly correlated with power consumption in ASH, this graph shows how much energy is spent for the extra performance.

For most programs the results are very good, the two bars being of comparable sizes. The worst outlier is the function `internal_filter` from `epic_e`, which has a huge IPC and exhibits a slowdown. An analysis of this function is presented in Section 8.2.2.3.

The most interesting result is for the function `FR4TR` `rasta`, which computes a Fourier Transform. This program has a huge speed-up obtained with a relatively low IPC increase. Interestingly enough, there are no pragma annotations either. So this is not the source of the increased efficiency of CASH. By analyzing the run-time statistics, we notice that the number of memory accesses of ASH in this function is an order of magnitude smaller than for the processor. Indeed, by disabling the inter-iteration register promotion algorithm described in Section 4.6, the performance of ASH drops by a factor of six! The rest of the speed-up is easily attributable to the higher IPC sustainable by ASH.

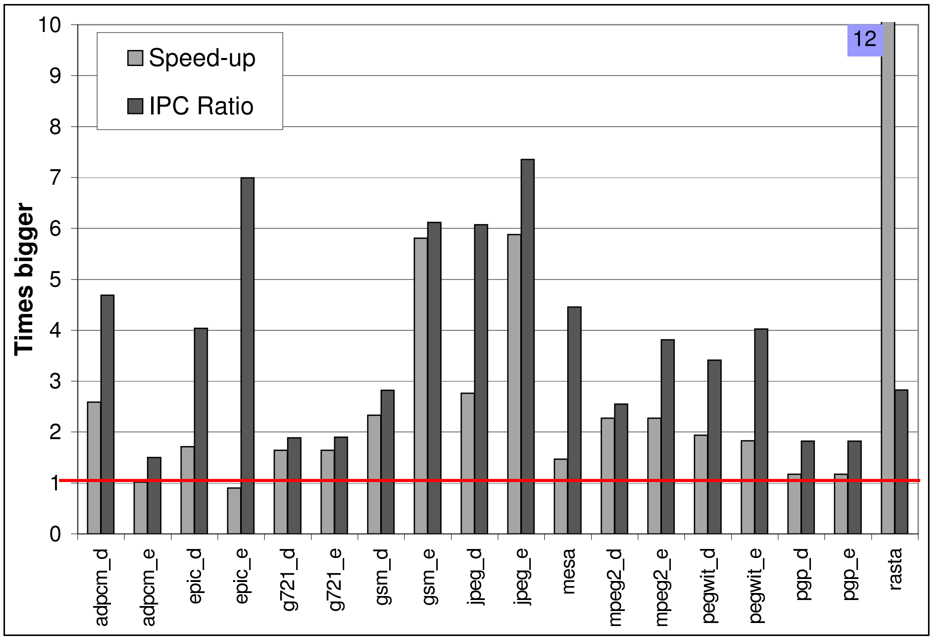


Figure 7.5: *Comparison of speed-up and IPC ratio.*

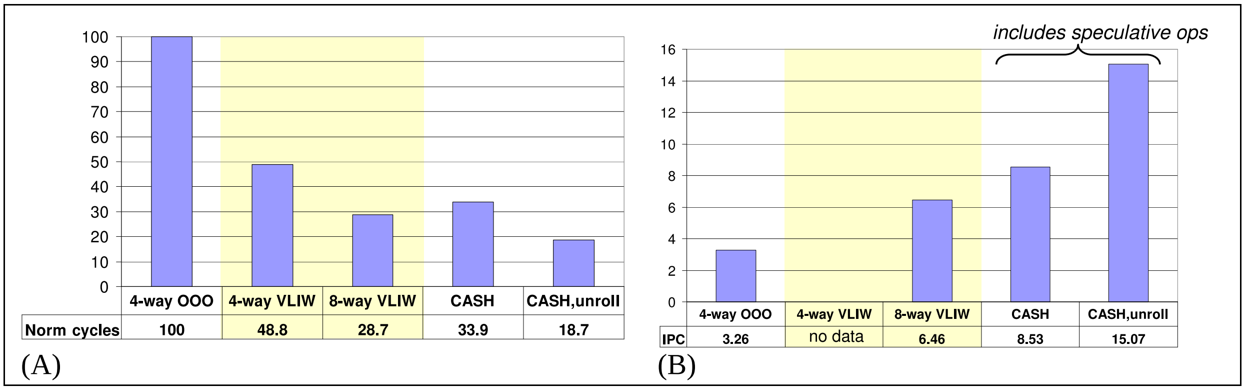


Figure 7.6: *Performance of the IDCT transform on five different run-time systems: (A) normalized execution cycles; (B) IPC.*

7.4.3 Comparison to a VLIW

We have attempted to use SimpleScalar as an approximation to an eight-wide VLIW processor. But the results are suspicious so we do not present them here (the sustained IPC was less than 1.1 for all programs, much too low). In this section we present a comparison of one benchmark we have obtained from Davide Rizzo and Osvaldo Colavin from ST Microelectronics, which was used to evaluate the performance of both (1) a very wide run-time reconfigurable clustered VLIW processor [RC03] on which code is mapped by hand and (2) the Hewlett-Packard Lx architecture [FBF$^+$00] with a sophisticated compiler based on Multiflow [LFK$^+$93]. The program is a fixed-point C version of the inverse discrete cosine transform (IDCT) used in MPEG movie encoding, taken from the OpenDIVX codec.

Figure 7.6 shows the results on this kernel as measured on five different platforms. The data for both

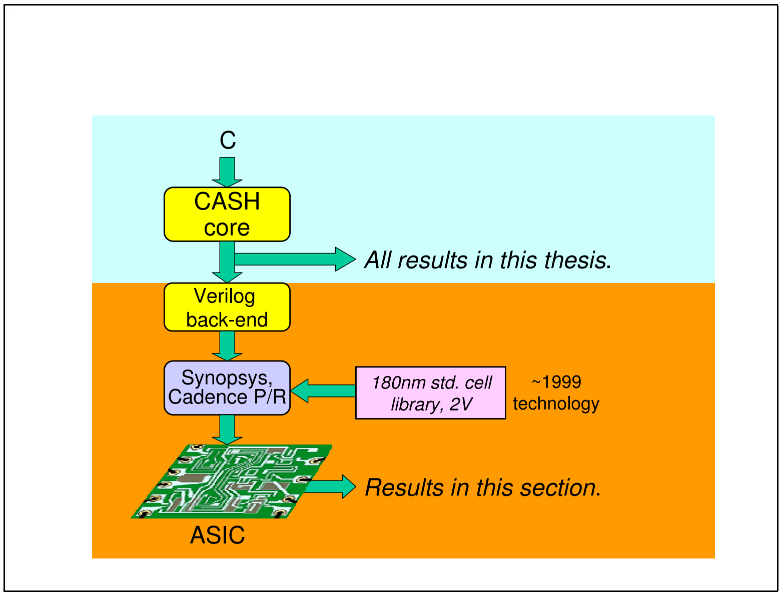


Figure 7.7: *Experimental set-up for performing the low-level measurements.*

VLIW engines is from ST. From left to right, we have:

1. Four-way out-of-order SimpleScalar with `gcc` 2.7..
2. A four-way Lx processor with a Multiflow-based compiler.
3. An eight-way reconfigurable VLIW, hand-compiled.
4. ASH+CASH without using any loop unrolling.
5. ASH+CASH with aggressive (eight times) loop unrolling.

Figure 7.6(A) shows the normalized execution time in “clock cycles”, assuming the same clock for all architectures. The operation latencies for the VLIW engines are different. The L1 access latency is the same in all experiments. The memory latency is different, but since this benchmark has very few cache misses it does not make much of a difference.

The ASH circuits generated with unrolling offer the best performance under these assumptions; ASH is 5.3 times faster than the superscalar and 50% faster than the eight-wide processor. Figure 7.6(B) shows the sustained IPC for four of the five platforms. ASH is somewhat more wasteful, retiring more instructions for obtaining a given level of performance.

7.5 All the Way to Verilog

In this section we present measurements from a detailed low-level simulation. Figure 7.7 shows the set-up used for performing the low-level measurements. A detailed description of our methodology can be found in [VBG03]. We use kernels from the Mediabench suite [LPMS97] to generate circuits. From each program we select one hot function (see Table 7.3) to implement in hardware (note that we are not using the exact same kernels as used in Section 7.4). The data we present is for the entire circuit synthesized by the asynchronous back-end, including the memory access network, but excluding the memory itself or I/O to the circuit. We report data only for the execution of the kernels itself, ignoring the rest of the program. We do not estimate the overhead of invoking and returning from the kernel. Since our current back-end does not support the synthesis of floating-point computation we had to omit some kernels, such as the ones from the `epic`, `rasta` and `mesa` benchmarks.

The CAB back-end is used to generate a Verilog representation of each kernel. A detailed description of our methodology can be found in [omitted]. We use a [180nm/2V] standard cell library from STMicroelectronics. The structural Verilog generated by our toolflow is partially technology-mapped by CAB and partially synthesized with Synopsys Design Compiler 2002.05-SP2. Simulation is performed with Modeltech Modelsim SE5.7. We assume a perfect L1 cache, and no LSQ for ASH; the baseline processor we use in our comparisons also uses a perfect L1 cache.[6] The power numbers are obtained from Synopsis from the switching activity on each wire measured during simulation.

The data in this document is obtained prior to actual place-and-route, and relies on statistical estimates about wire lengths. Bugs in the tool-chain have prevented us from successfully measuring all programs post-layout. However, the data we have obtained shows that results improve by about 15% after layout, both in terms of performance and power, with a minor penalty in area.

The most difficult aspect of synthesis is building the memory access network for the load and store operations. The network is synthesized using some asynchronous pipelined arbiters to mediate access to a single memory port.

This evaluation differs in some significant respects from the other measurements presented in this document:

- The Verilog back-end does not yet handle procedure calls and returns. Therefore, all kernels consist of a single function[7].

- The back-end uses a separate "fanout" node to implement computations with fanout; all other operations have strictly one consumer. This will increase the critical path of most circuits.

- Since our current back-end does not support the synthesis of floating-point computation we had to omit some kernels, such as the ones from the `epic`, `rasta` and `mesa` benchmarks.

- The memory access protocol currently implemented is very inefficient: in order to ensure in-order execution of memory operations, an operation does not release its token until it has reached memory and performed its side effect. A memory request has thus to traverse the memory network back and forth completely before dependent operations may be initiated.

- Memory operations are initiated only when all inputs are available.

- We do not use pipeline balancing. This optimization depends on accurate operation latencies to compute the period of each cycle, which are not known until actual hardware synthesis.

- We do not currently pipeline any operation.

[6] These assumptions are not completely unreasonable for some classes of embedded processors.

[7] For `g721` we have inlined the function `quan` in `fmult`, manually unrolled the loop and inlined the constant values read from a memory array. The last optimization could not be performed by our compiler, since it requires an inter-procedural side-effect analysis. The baseline superscalar uses the same code as us.

Benchmark	**Function**
adpcm_d	adpcm_decoder
adpcm_e	adpcm_coder
g721_d	fmult+quan
g721_e	fmult+quan
gsm_d	Short_term_synthesis_filtering
gsm_e	Short_term_analysis_filtering
jpeg_d	jpeg_idct_islow
jpeg_e	jpeg_idct_fslow
mpeg2_d	idctcol
mpeg2_e	dist1
pegwit_d	squareDecrypt
pegwit_e	squareEncrypt

Table 7.3: *Embedded benchmark kernels selected for the low-level measurements.*

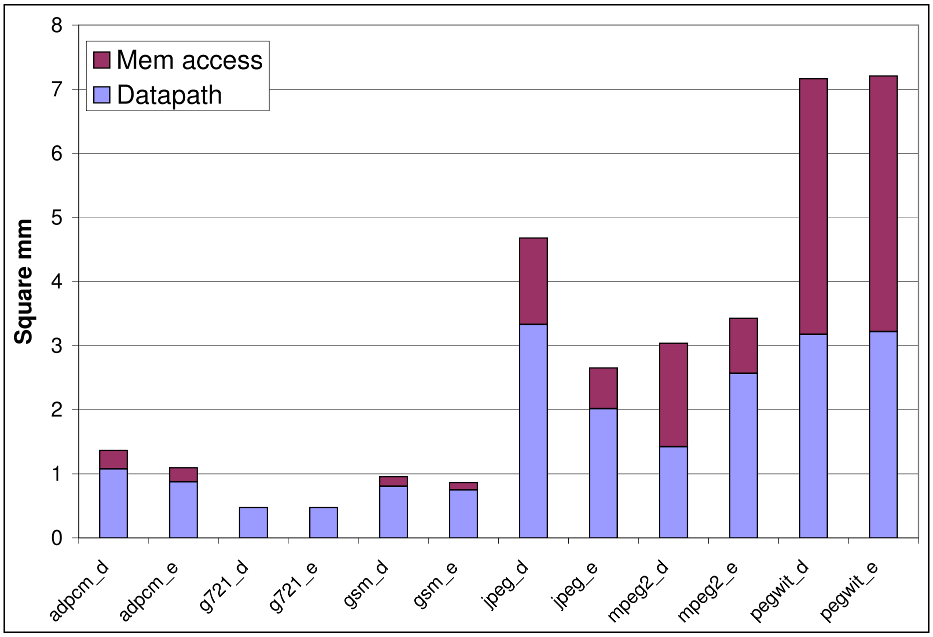


Figure 7.8: *Silicon real-estate in mm^2.*

- This data does not reflect the effects of the BitValue data width analysis, which at the point of this writing wasn't yet completely integrated within the tool-flow. We expect reductions on the order of 20%, and as high as 50%, of the area and power drawn by the datapath once this algorithm is integrated.

- We do not use loop unrolling. We expect this will increase performance at the expense of area.

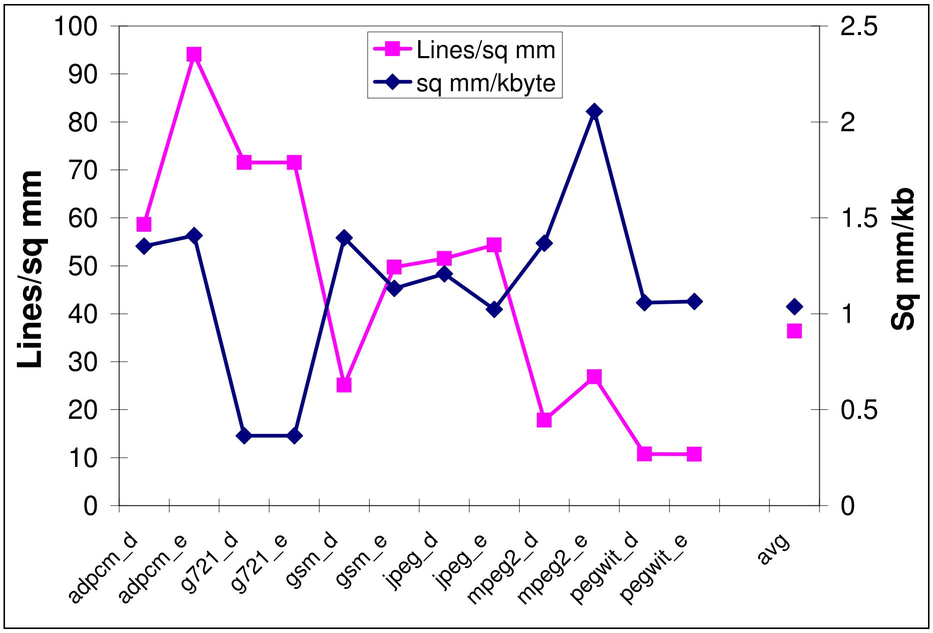


Figure 7.9: *Normalized area: source lines per mm^2 and mm^2 per kilobyte of MIPS text-file object file.*

7.5.1 Area

Figure 7.8 shows the amount of silicon real-estate required for the complete implementation of these kernels. The area is broken down into "computation" and "memory tree." The memory tree is the area of the arbiter circuits used to mediate access to the memory load-store queue.

For reference, in the same technology a minimal RISC core can be synthesized in 1.3mm^2, a 16 × 16 multiplier requires 0.1mm^2, and a complete P4 processor die, including all caches, has 217mm^2. This shows that while the area of our kernels is sometimes substantial, it is certainly affordable, especially in future technologies.

Figure 7.9 shows the area normalized using two different metrics: lines of source code and object file size. The source-code metric is particularly sensitive to coding style, (e.g., macro usage); we measure source lines before preprocessing, including whitespace and comments. Normalizing the area versus the object file size indicates that we require on average 1mm^2 for each kylobyte of a gcc-generated MIPS object file.

7.5.2 Execution performance

Figure 7.10 shows the normalized execution time of each kernel having as a baseline a 600MHz[8] 4-wide superscalar processor[9]. On average 2.05 times slower. While the processor computational units are expected to be faster than ASH's, the large amount of parallelism exploitable by ASH makes these results somewhat disappointing. Our analysis has uncovered two major sources of inefficiency, which we are targeting in our future work: the C-elements in our designs and the memory access network.

1. C-elements [MB59] are frequently used in asynchronous design for implementing control. Unfortu-

[8] This is a high speed for our particular technology.

[9] We did not have access to an accurate VLIW simulator; we expect the trends to be similar, since the superscalar maintains a high IPC for these kernels.

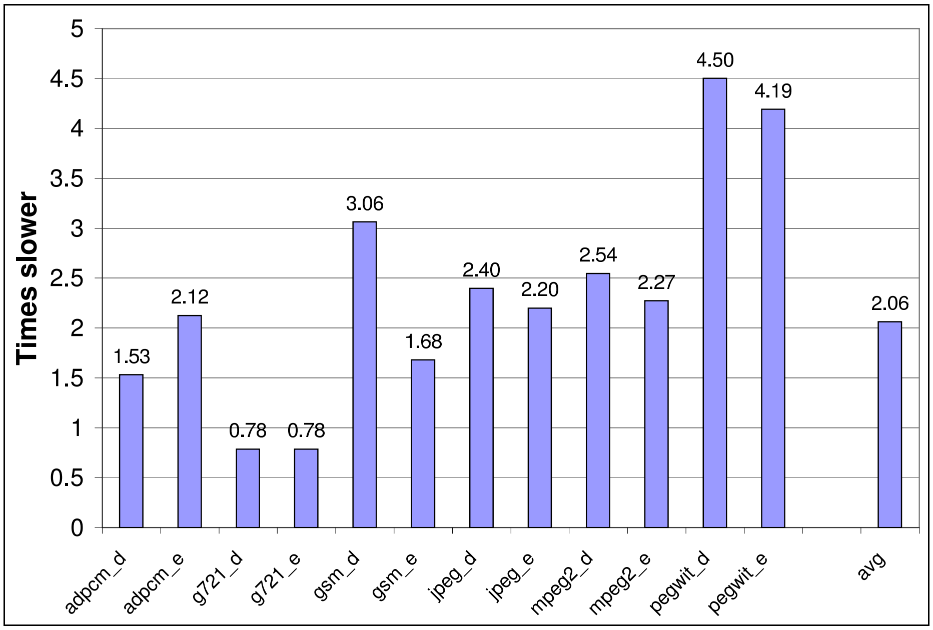


Figure 7.10: *Kernel slowdown compared to a 4-wide issue 600Mhz superscalar processor in 180nm.*

natelly, the standard gate library used for our current experiments does not contain such a gate; using this library the smallest implementation of a C-element consists of two gates, one of which is quite complex. In 0.18μm technology we estimate that a custom C-element would be 66% faster and 75% smaller. When simulating with a behavioral implementation of such a gate the end-to-end execution latency of our kernels improves by 25%. We are currently augmenting the library with a custom C-element.

2. In our current implementation a memory operation does not release a token until it has traversed the network end-to-end in both directions. An improved construction would allow an operation to (1) inject requests in the network and (2) release the token to the dependent operations immediately. The network packet can carry enough information to enable the memory to execute the memory operations in the original program order. This kind of protocol is actually used by superscalar processors, which inject requests in order in the load-store queue, and can proceed to issue more memory operations before the previous ones have completed.

   To gauge the impact of the memory network on program performance we perform a limit study using a behavioral implementation which reduces the cost of an arbitration stage to 10ps (a close approximation to zero). The limit study indicates that with a very fast network most programs exceed the processor in performance; in particular, the slowest kernels, `pegwit`, speed-up by a factor of 6. This indicates that our programs are bound by the memory network round-trip latency. The current communication protocol should be the first optimization target.

In Figure 7.11 we present the program performance for the actual technology-mapped implementation using several MIPS metrics: the bottom bar we labeled MOPS, for millions of useful arithmetic operations per second. The incorrectly speculative arithmetic is accounted for as MOPSspec. Finally, MOPSall includes “auxiliary” operations, including the merge, eta, mux, combine, and other overhead operations. Although speculative execution sometimes dominates useful work (e.g., `g721`), on average 1/3 of the executed

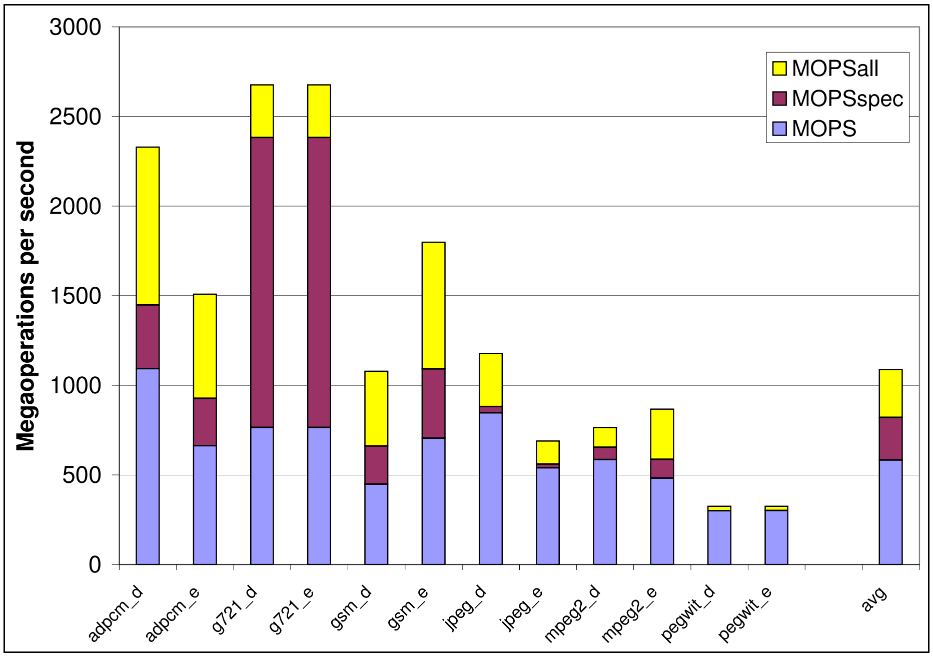


Figure 7.11: *Computational performance of the ASH Mediabench kernels, expressed in millions of operations per second (MOPS).*

arithmetic operations are incorrectly speculated.[10] Most programs sustain half an effective GigaOP/sec.

In order to accurately estimate the number of speculative instructions executed, the run-time system logs the predicates controlling muxes, etas and predicated operations. The compiler is then invoked to parse the log and to perform a backward dataflow analysis, discovering all operations whose output was "useful." Each hyperblock execution requires a separate dataflow analysis. Due to very long simulation times of the Verilog circuits, this data reflects only the first execution of each kernel (each kernel may be invoked repeatedly for a program).

Since the circuits are asynchronous, we cannot speak of a "clock cycle", or estimate the performance with variation of the clock. We did not attempt to vary the supply voltage, which in asynchronous logic is one of the main factors that can be used to trade-off energy and performance.

7.5.3 Power and Energy

The power consumption of our circuits ranges between 7.4mW (gsm_d) and 32.8mW (pegwit_e), with an average of 15.6mW. In contrast, a very low power DSP chip in the same technology draws about 110mW, and a very low power superscalar about 4W.

In order to quantify power and energy efficiency we use the normalized metric proposed by Claasen [Cla99], MOPS/mW. This metric is the same as the number of operations per nanoJoule. Figure 7.13 shows how our kernels fare from this point of view. We count only the useful operations, i.e., the bottom bar from Figure 7.11. Therefore, these results are the most pessimistic estimate.

For comparison Figure 7.15 shows the energy efficiency of a set of microprocessors, digital signal processing chips and custom hardware circuits, extracted from [ZB02]. The chips are listed in Figure 7.14.

[10] The use of profiling information could be used to reduce the amount of incorrect speculation.

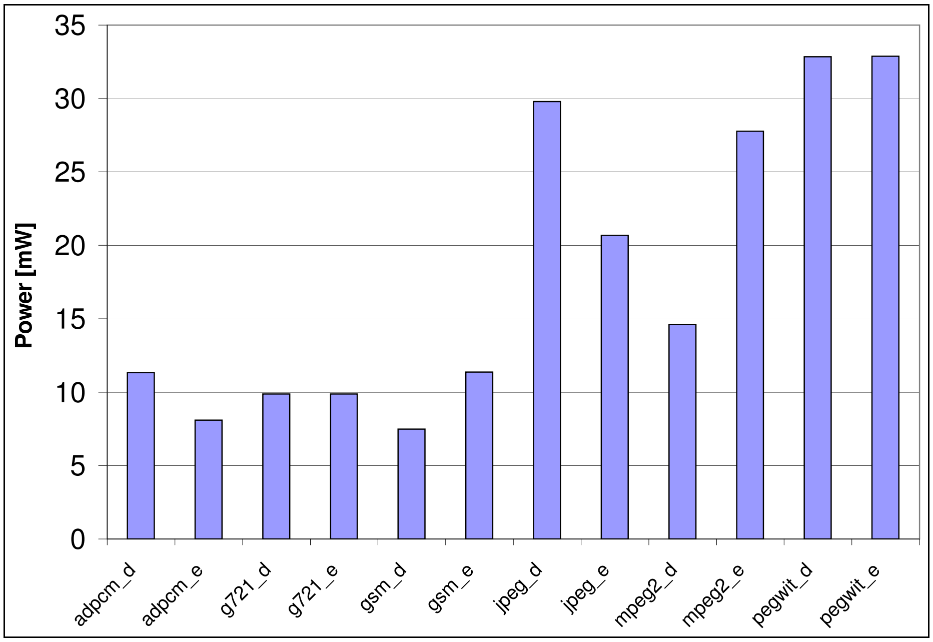


Figure 7.12: *Power consumption of kernels in mW.*

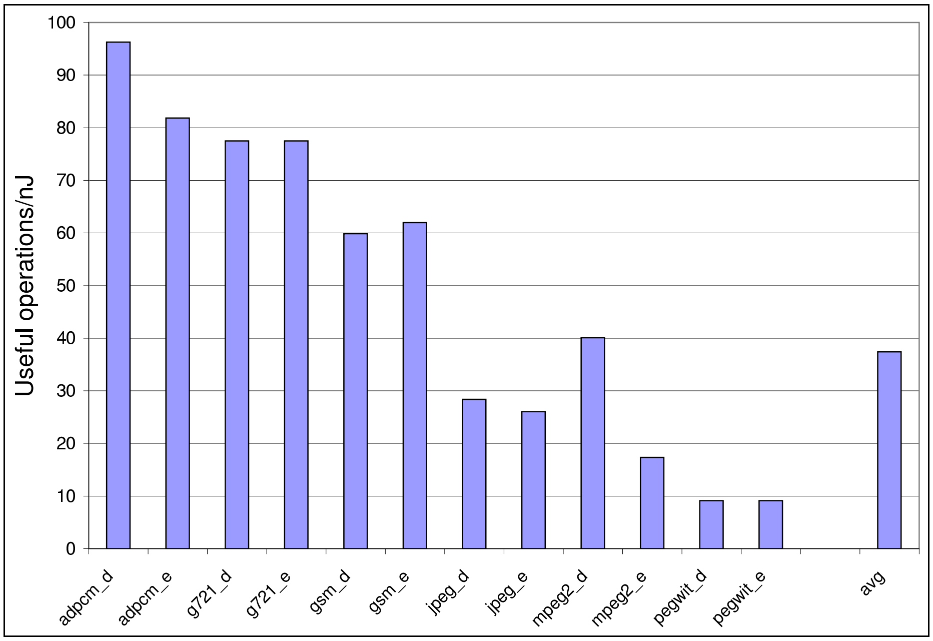
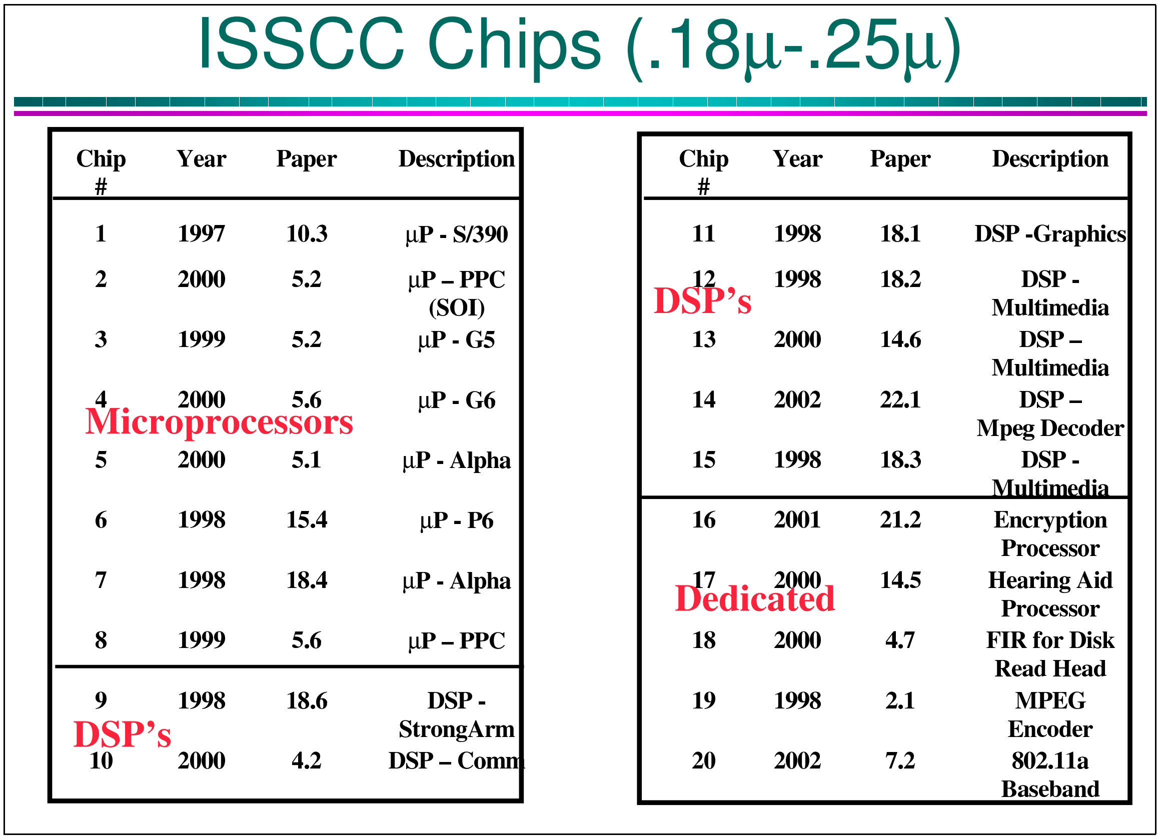


Figure 7.13: *Energy efficiency of the synthesized kernels, expressed in operations per nanoJoule, or millions of operations per second per milliwatt.*



ISSCC Chips (.18µ-.25µ)

Microprocessors

Chip #	Year	Paper	Description
1	1997	10.3	µP - S/390
2	2000	5.2	µP – PPC (SOI)
3	1999	5.2	µP - G5
4	2000	5.6	µP - G6
5	2000	5.1	µP - Alpha
6	1998	15.4	µP - P6
7	1998	18.4	µP - Alpha
8	1999	5.6	µP – PPC

DSP's

Chip #	Year	Paper	Description
9	1998	18.6	DSP - StrongArm
10	2000	4.2	DSP – Comm

DSP's

Chip #	Year	Paper	Description
11	1998	18.1	DSP -Graphics
12	1998	18.2	DSP - Multimedia
13	2000	14.6	DSP – Multimedia
14	2002	22.1	DSP – Mpeg Decoder
15	1998	18.3	DSP - Multimedia

Dedicated

Chip #	Year	Paper	Description
16	2001	21.2	Encryption Processor
17	2000	14.5	Hearing Aid Processor
18	2000	4.7	FIR for Disk Read Head
19	1998	2.1	MPEG Encoder
20	2002	7.2	802.11a Baseband

Figure 7.14: *20 chips compared with ASH: eight processors, seven DSPs, and five custom hardware chips. This figure is a reproduction of a slide by Bob Brodersen of Berkeley.*

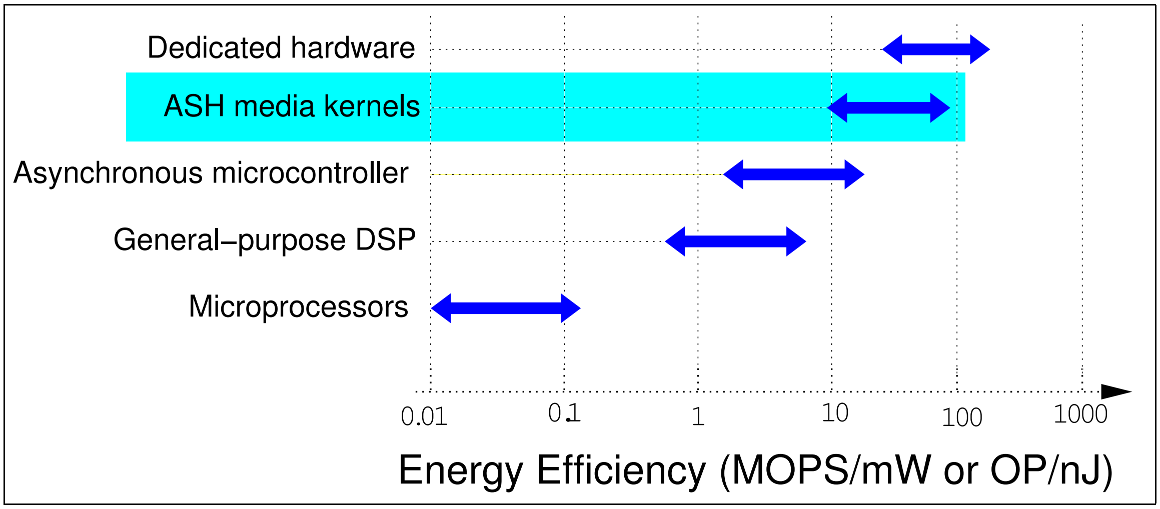


Figure 7.15: *Energy efficiency of the 20 chips from Figure 7.14, of an asynchronous microprocessor and of ASH. ASH is competitive with the custom hardware in energy efficiency. This is an adaptation of a slide by Bob Brodersen.*

We added to the comparison Lutonium [MNP$^+$03], a modern asynchronous microprocessor. Its efficiency varies with the supply voltage, trading-off performance for energy. At 1.8V and maximum performance of about 200 MIPS, Lutonium fares at 1.8 MOPS/mW, and at a low voltage of 0.5V, the performance is four MIPS at an efficiency of 23 MOPS/mW.

ASH fares three orders of magnitude better than a generic microprocessor and between one and two orders of magnitude better than a DSP. Data in [MNP$^+$03] suggests that more than 70% of the power of the asynchronous Lutonium processor is spent in instruction fetch and decode, operations which do not exist in ASH. This partially explains the higher efficiency of ASH.

7.5.4 Impact of Future Technologies

We expect that even our results using the 180nm technology can be substantially improved, by using both circuit-level techniques and high-level compiler optimizations. We are confident we can bring the slowdown very close to 1 or better, while not compromising power and area.

We expect that all of these results will scale very well with future technologies: speed-up should increase while area and power consumption should continue to decrease at the same rate as in synchronous designs. The impact of leakage power remains to be estimated, but we expect that high-level techniques, such as power gating at the hyperblock/function level under compiler control can be used to diminish its impact.

7.6 Conclusions

Although we have not presented supporting data here, both loop unrolling and memory disambiguation through insertion of `pragma` statements are essential for good performance. Both these tasks can be mostly automated: unrolling can be attempted by various factors under user control and the discovery of the most useful independent pointer pairs can be made using compiler-based instrumentation, as shown in [KBVG03].

The space of embedded computation is seemingly an ideal domain for Spatial Computation: the applications have small compute-intensive kernels exhibiting an important amount of data parallelism in the form of IPC and pipeline parallelism, which CASH takes advantage of naturally. The modest size of these kernels makes them suitable for hardware-based implementation. Low-level simulations suggest that both the sustained performance and energy efficiency of ASH are excellent.

Chapter 8

Spatial Computation and Superscalar Processors

This chapter is devoted to a comparison of the properties of Spatial Computation and superscalar processors. The comparison is carried by executing whole programs on both computational fabrics and evaluating the performance of the resulting implementation. The most important result of this study is a qualitative comparison of the intrinsic computational capabilities of Spatial Computation and superscalar processors.

8.1 Introduction

The comparison we perform here is closer to a limit study of the capabilities of ASH than to an accurate evaluation, due to the large number of parameters with uncertain values involved. This comparison contrasts the performance potential of unlimited instruction-level parallelism against a more sophisticated approach involving prediction and speculation. Another way to characterize this study is: a comparison of static dataflow against a resource-limited dynamic dataflow with speculation.

Before proceeding with the comparison, let us enumerate the expected advantages of each model of computation. We consider not only performance-related aspects but also economic, engineering, correctness, etc. implications. Whether ASH is a feasible model of computation will hinge on a combination of all these factors.

8.1.1 Spatial Computation Strengths

Application-specific: The biggest luxury of ASH designs is that they are created *after* the application is known. This is why ASH has no notion of instruction and none of the baggage associated with instruction handling: fetch, decode, dispatch.

Maximal parallelism: this is a direct consequence of the application-specific nature. Since the executed program is known a priori, the compiler provisions exactly as many resources as necessary to carry that computation. In Spatial Computation there are no resources hazards[1] and all ILP limitations stem from the compiler.

Lean hardware: As we have argued in Section 7.1, the computational structures in ASH are very simple, requiring plain arithmetic units connected by unidirectional communication links and a very simple

[1] Strictly speaking, both memory and LSQ bandwidth are limited resources and may give rise to hazards.

handshaking protocol. The interconnection network is very simple in contrast to the complex data forwarding paths inside of a typical processor pipeline, or the very complex control circuitry for handling speculation and resource arbitration. We expect the lean hardware to translate into the ability of synthesizing large programs with high computational speeds and low power consumption.

Dynamic scheduling: Unlike VLIW engines, ASH does not rely on complete control over scheduling at compilation time. Therefore, it can absorb unpredictable latency events, stemming either from remote operations such as procedure calls and memory accesses, or simply from variability in the layout of the final circuit as processed by the back-end tools.

No fixed ISA: The program computation does not have to be expressed as a composition of a set of fixed primitives. While our compilation infrastructure does not yet take full advantage of this freedom, CASH does generate constructs which do not correspond to instructions in a regular ISA, such as the token edges.

No finite instruction window: In Spatial Computation there are virtually as many "program counters" as there are "live" data values. New instructions are issued as soon as enough inputs are present. There is also no resource limitation on the number of active instructions and no "in-order" commit requirements.

Simpler validation: Although it is somewhat premature to make any claims on the ease of certifying Spatial Computation designs, it is clear that they eliminate an important piece of the trusted computing base, the processor. We foresee, as an important piece of future research, an effort for the automatic formal validation of the compilation process. This could be carried out either as translation validation [PSS98, Nec00], which can prove the equivalence of the initial CFG and the optimized Pegasus form (or perhaps just of the unoptimized and optimized Pegasus), or a certification of the correctness of the compiler transformations. The asynchronous nature of the generated circuits should contribute to the ease of verification through the modularity and composability intrinsic of this model of computation.

8.1.2 Superscalar Processor Strengths

We choose to compare ASH against a superscalar processor. While superscalars have some well-known scalability problems, such as high power consumption and huge design complexity, they are the appropriate opponent to compare against because their internal dataflow engine is closely related to ASH's dynamic scheduling. In this section we enumerate the strengths of the superscalar and contrast them to weaknesses of ASH.

Mature technology: A substantial expertise exists in developing both synchronous circuits and microprocessors of all kinds.

Universal: Unlike ASH, the processor is a universal device which, once built, can run any program. This justifies an increased investment in designing and optimizing it. For example, each of the (constant number) of arithmetic units is usually optimized for speed at the expense of area, a trade-off that in general is not acceptable for ASH.

Economies of scale: Despite the huge cost of manufacturing plants, design and verification, the final cost of processors is quite small (especially embedded or older generations), due to the enormous economies of scale.

Virtualizable: Once the ISA is sufficiently rich, the microprocessor can easily be used to virtualize resources by providing multiprocessing and time-sharing, exception and interrupt handling, virtual memory, etc.

Flexible: Microprocessors can be used for purposes not initially envisioned. For example, they allow dynamic (run-time) optimizations.

Resource sharing: The same relatively small number of arithmetic units is reused for executing all program instructions.

Full dataflow: Unlike ASH in a superscalar processor multiple instances of the same static instruction can be in flight at the same time, due to register renaming.

Monolithic: While the monolithic structure of a processor is the most important stumbling block for scaling the architecture and validating a design, it has some very important benefits — such as allowing the use of *global dynamic optimizations*. These are mechanisms that can collect information from various points of the pipeline and use it to make global decisions. Two important examples, as shown in Section 8.2.2, are (1) global branch prediction and (2) speculative execution — the monolithic construction allows a coordinated flushing of the pipeline on detection of mis-speculation.

8.2 ASH versus SuperScalar

In this section we compile whole programs using CASH and evaluate their potential performance. We compare to a superscalar processor baseline. We assume the same operation latencies as shown in Table 7.1 from Chapter 7.

Some of these assumptions are clearly optimistic for ASH when dealing with whole programs. In particular, the complexity of the memory access network and of the interconnection medium for procedure calls and returns cannot be assumed constant anymore. In the best case the latency of these networks should grow as $\sqrt{n}$, where n is the size of the compiled program, assuming a two-dimensional layout. Therefore, one should interpret the data in this section more as a limit study than as absolute magnitudes.

8.2.1 Performance Measurements

Figure 8.1 shows the speed-up of complete applications from SpecInt95 executed under ASH when compared to a superscalar processor. Unfortunately most of the numbers indicate a slowdown. Only `099.go`, which is very array-intensive, and `132.ijpeg`, which is very similar to the Mediabench `jpeg` programs, show some performance improvements. Results for SpecInt2K show similar trends and are not presented due to excessive simulation time requirements.

8.2.2 Critical Code Segments

We would like to better understand the results presented above. Naive expectations are that the performance of ASH cannot be surpassed since it benefits from (a) unlimited parallelism, (b) lack of computational resource constraints, and (c) dynamic scheduling. However, we have already hinted at some unique capabilities of a superscalar core in the introduction of this chapter. In this section we display a few functions and code fragments which exhibit limitations of ASH. It is certainly possible that there are other handicaps to ASH, other than the ones we enumerate here. However, the investigation process is mostly manual and quite tedious, so it is an ongoing process.

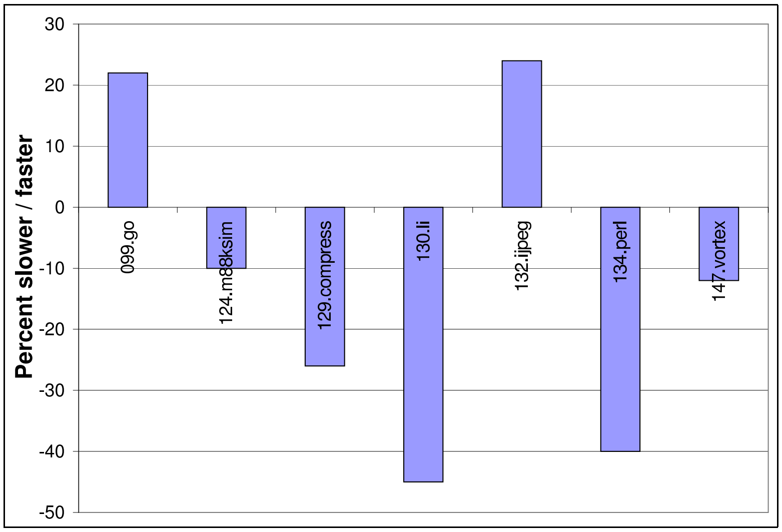


Figure 8.1: *Speed-up of SpecInt95 benchmarks executed on ASH. The baseline is a superscalar four-way out-of-order processor. Negative values indicate a slow-down.*

8.2.2.1 `epic_d:internal_int_transpose`

The code for this function is shown verbatim in Figure 8.2. The performance on ASH is about two thirds of the performance on the superscalar. There are three branches involved in two very shallow nested loops: (1) the `do` backwards branch, mostly not taken (this loop iterates either once or twice); (2) the `if` branch, almost always taken; and (3) the `for` backwards branch, almost always taken. There are very few machine instructions in the outer loop body (eight to be precise) and the processor window is large enough to hold several iterations worth. By using branch prediction the processor can effectively issue a second iteration of the *outermost* `for` loop before having completed the previous one. The iterations of the `for` are effectively pipelined. The critical resource in this loop is the single division unit, which is not pipelined, and which takes 24 cycles to complete a modulus. Indeed, the throughput of this function averages 27 cycles per outer loop iteration. The cost of most operations (multiply, branches, memory accesses) is hidden by the division.

For the ASH implementation the most important part of the critical path of this function, highlighted using the tools described in Section 6.6, is shown in bold lines in Figure 8.3. The performance is hampered by the strict sequencing of control along the three hyperblocks. In particular, a new iteration of the outer loop cannot be initiated as long as the inner loop has not terminated. This also puts the three-cycle multiplier on the critical path. Counting the extra operations on this path (besides the modulo) we get 12 extra cycles, which corresponds indeed with a 33% slowdown. The predicates controlling both the loops, [66] for the `do` and [213] for the `for`, are on the critical path. [66] is the “last” value computed within the innermost loop and it is immediately used to decide whether to loop again. The case for [213] is slightly different: although the value is available much earlier and is unchanged within the `do` loop, it is not *propagated* out of the [83][228] loop until this loop’s iterations are completed, as indicated by the loop termination predicate [70].

A simple optimization could completely bypass the innermost loop for the `current_pos` value. However, such an optimization can lead to a violation of the program invariants if the innermost loop never terminates — not all operations within the bottom hyperblock would execute. We are currently planning to implement this optimization, either only for provably terminating loops or for all loops if directed by

```
/*
=============================================
In-place (integer) matrix tranpose algorithm.
=============================================
*/
internal_int_transpose( mat, rows, cols )
  register int *mat;
  register int cols;
  int rows;
  {
  register int modulus = rows*cols - 1;
  register int swap_pos, current_pos, swap_val;

  /* loop, ignoring first and last elements */
  for (current_pos=1; current_pos<modulus; current_pos++)
      {
      /* Compute swap position */
      swap_pos = current_pos;
      do
          {
          swap_pos = (swap_pos * cols) % modulus;
          }
      while (swap_pos < current_pos);

      if ( current_pos != swap_pos )
          {
          swap_val = mat[swap_pos];
          mat[swap_pos] = mat[current_pos];
          mat[current_pos] = swap_val;
          }
      }
  }
```

Figure 8.2: *The `internal_int_transpose` function from the `epic_d` benchmark from Mediabench, which exhibits poor performance on ASH.*

compilation options.

8.2.2.2 `124.m88ksim:init_processor`

Let us now analyze another piece of code which shows poor performance on ASH, shown in Figure 8.4. ASH takes more than twice the number of cycles of the processor to execute this code. Figure 8.5 shows the essential part of the code generated by ASH with the critical path highlighted. Despite the brevity of the code, quite a few optimizations have been applied: the load from `SFU0bits` has been register-promoted, the address computation of `SFU0bits[j].regnum` which involved a multiplication by 20 (the size of the structure `SFU0bits[0]`), has been strength-reduced (it is computed by the loop [316][277][275]). As in the case of the previous function, the predicate deciding whether the loop should iterate once more is the

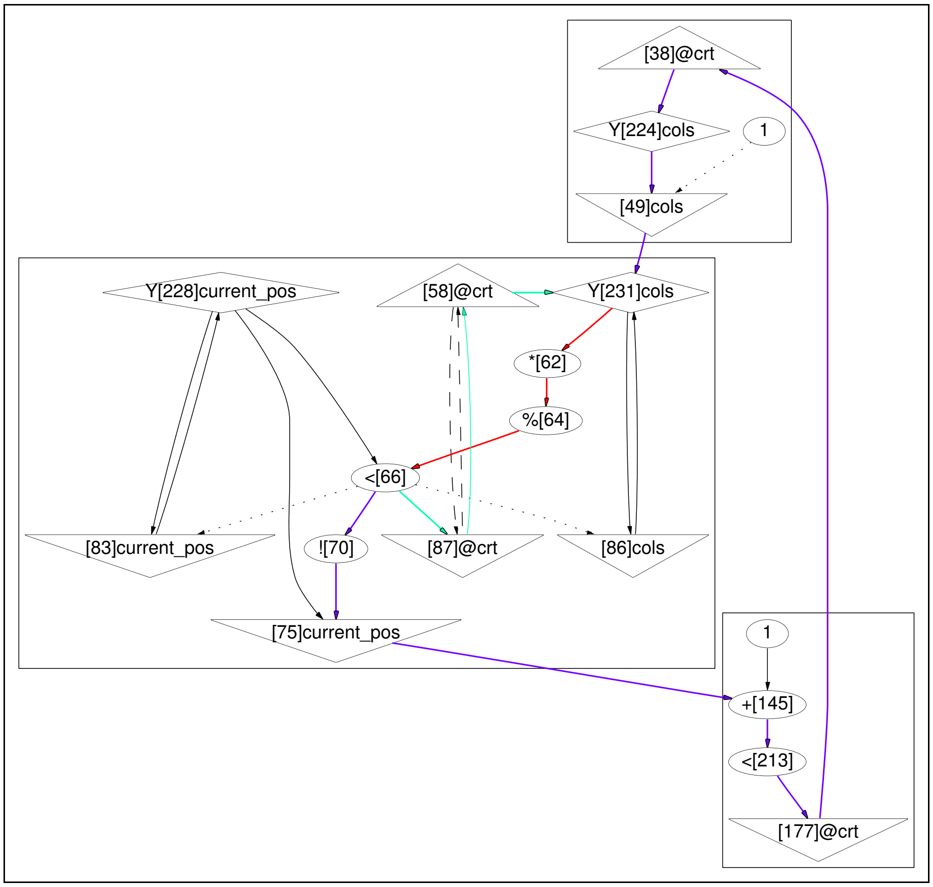


Figure 8.3: *Dynamic critical path of the function* `internal_int_transpose` *in ASH. The critical path alternates between the innermost cycle [231]→[62]→[64]→[66]→[87]→[58]→[231] and the outer loop [38]→[224]→[49]→[231]→[63]→[64]→[66]→[70]→[75]→[145]→[213]→[177]→[38]. The edge [87]→[58] on the critical path is the acknowledgment edge corresponding to [58]→[87].*

last thing computed because it depends on the loaded value. The critical path contains not only the load but the load predicate computation as well, since it indicates whether the load has to be executed or not.

The LSQ may be configured to initiate a load that reaches the head of the queue speculatively, i.e., without waiting for the actual predicate value. However, doing so changes only the trajectory of the critical path, but not its length — now it goes through the address computation [50][316][331][76]. The problem is that the address computation for the next iteration is not made speculatively — it waits for a signal from the [50] *crt* node to initiate the computation.

The superscalar code has about the same shape and the same optimizations are applied. Strength reduction removes the multiplier and PRE the repeated load. It is shown in Figure 8.6. There is a loop-carried dependence involving four of the five instructions ($\mathrm{L1}\xrightarrow{control}\mathrm{L2}\xrightarrow{true}\mathrm{L3}\xrightarrow{true}\mathrm{L5}\xrightarrow{control}\mathrm{L1}$), so the latency through this loop ought to be at least four cycles. However, if branch prediction correctly guesses the outcome of the two branches, the control dependences vanish and the loop-carried dependence becomes much shorter. The only remaining dependences are between L2 and L3. The processor can execute this loop in only two cycles per iteration!

Now, let us assume that we implement branch prediction in Pegasus and therefore we can initiate a

```
void init_processor(void)
{
    int i, j;

    for(i = 0; i < 64; i++)
    {
        for(j = 0; SFU0bits[j].regnum != 0xFFFFFFFF; j++)
            if(SFU0bits[j].regnum == i)
                break;

        m88000.SFU0_regs[i] = SFU0bits[j].reset;
    }

    /* another almost identical loop removed */
}
```

Figure 8.4: *Half of the code from the* `init_processor` *function from the* `124.m88ksim` *benchmark from SpecInt95, which exhibits poor performance on ASH.*

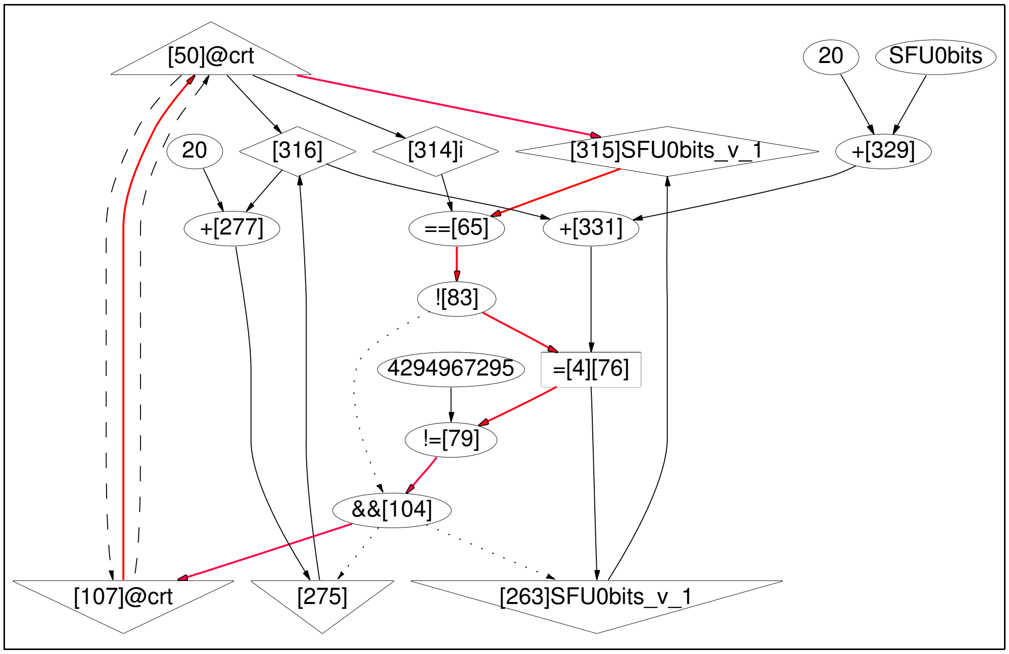


Figure 8.5: *Dynamic critical path of the function* `init_processor` *in ASH. The most frequent edges are [50]→[314]→[65]→[83]→[76]→[79]→[104]→[107]→[50].*

```
LOOP:
L1:   beq $v0,$a1,EXIT     /* SFU0bits[j] == i */
L2:   addiu $v1,$v1,20     /* &SFU0bits[j+1] */
L3:   lw $v0,0($v1)        /* SFU0bits[j+1] */
L4:   addiu $a0,$a0,1      /* j++ */
L5:   bne $v0,$a3,LOOP     /* SFU0bits[j+1] == 0xffffffff */
EXIT:
```

Figure 8.6: *Assembly code for the innermost loop of the function in Figure 8.4.*

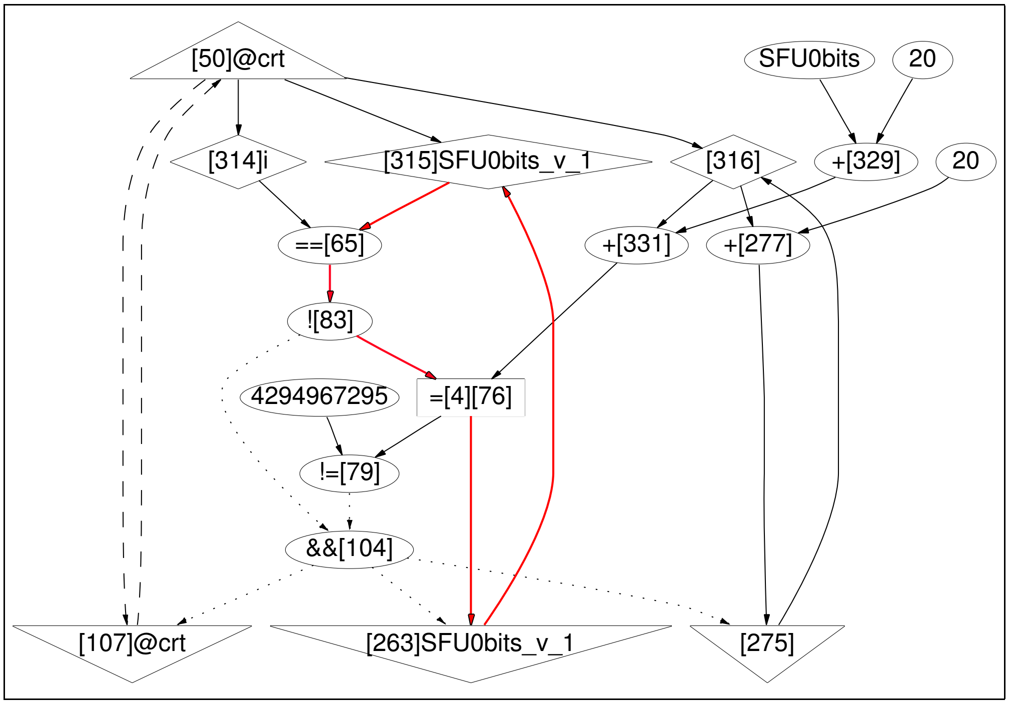


Figure 8.7: *Dynamic critical path of the function* `init_processor` *in ASH when using perfect eta predicate prediction. The most frequent edges are [315]→[65]→[83]→[76]→[263]→[315].*

new iteration early, without waiting for the predicates to be known. Do we achieve the same throughput? Section 8.3 describes how we carried out such a study. With perfect prediction the performance of the ASH code for this procedure increases, but still does not match the processor.

Figure 8.7 shows the critical path when the value of [104] is available instantaneously. Under the assumption that the load is not executed speculatively, the path no longer contains the *crt* nodes but still contains the load predicate, which prevents the load from being issued to memory.

The memory protocol we have described in Section 6.5.2 allows loads that reach the head of the LSQ and still have an unknown predicate to be issued to memory (i.e., to execute speculatively). If we draw the critical path under this last assumption, we obtain Figure 8.8.

This last critical path contains an acknowledgment edge, from the load to its address computation. Technically the limiting dependence is an anti-dependence between the output register of the load in two

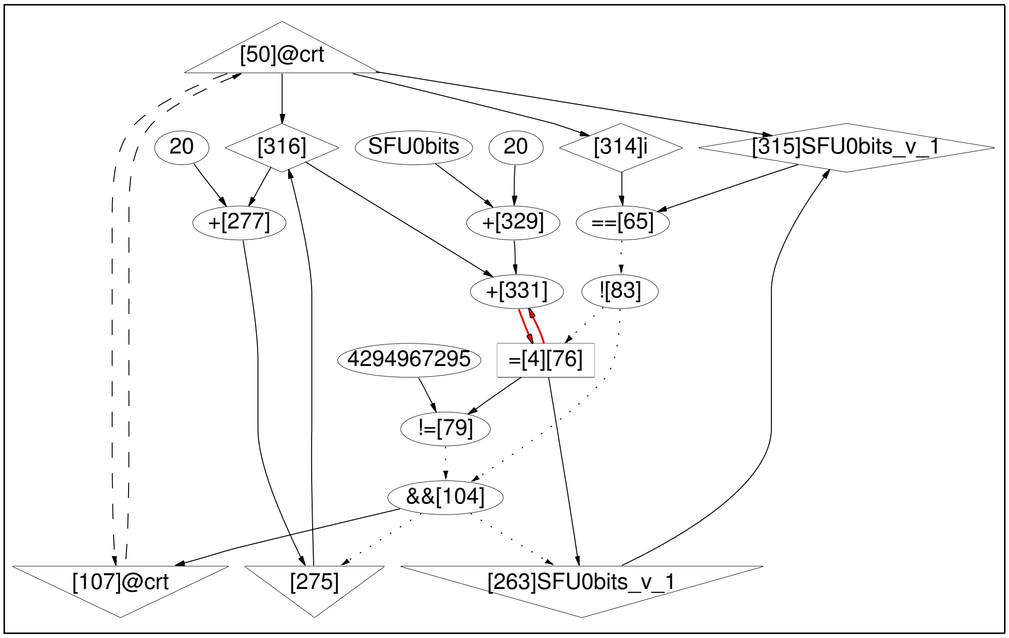


Figure 8.8: *Dynamic critical path of the function* `init_processor` *in ASH when using perfect eta predicate prediction and speculative loads. The most frequent edges are [331]→[76]→[331], including an acknowledgment edge.*

different iterations. This dependence in ASH stems from the fact that the load has a single output register, and therefore cannot acknowledge its address input before it has successfully latched its result. However, despite hitting in the cache, the latency of the load is more than two cycles — it takes one cycle just to get to the LSQ, and then two more cycles to hit into L1. As we pointed out in Section 6.7.2.2, the rate of a computation is *bounded below* by the longest latency operation. If a load takes three cycles, there is no way the computation can complete in only two.

We can imagine two solutions for this problem:

1. Loop unrolling, which transforms one load into multiple loads made in parallel. The latency of the computation is still bounded below by the latency of a load. But the limit now applies to multiple iterations of the original loop.

2. Pipelining the load operation itself to allow multiple simultaneous accesses initiated by the same operation. This solution requires complicated hardware for implementing a load, which must be able to buffer and handle requests out-of-order.

Let us notice that a superscalar processor removes the anti-dependence on the result of the load by register renaming. It can therefore have multiple instances of the same load operation in flight, without any problem whatsoever. This proves that a dynamic dataflow model, which is well approximated by a Superscalar when enough computational resources are present, is strictly more powerful than a static model — the one employed in Spatial Computation.

Finally, unrolling the `init_processor` loop and using perfect branch prediction gets closer to the processor performance, but is still not a match. The reason is because the dynamic size of the inner loop varies, having sometimes many and sometimes few iterations. Unrolling helps when the number of iterations is large but leads to pure overhead when the number of iterations is smaller than the unrolling factor.

```
double sum;
register float * RESTRICT image;

for (...) {
    for (...) {
        ...
        sum=0.0;
        for (y_filt_lin=x_fdim,x_filt=y_im_lin=0;
             y_filt_lin<=filt_size;
             y_im_lin+=x_dim,y_filt_lin+=x_fdim)
          for (im_pos=y_im_lin;
               x_filt<y_filt_lin;
               x_filt++,im_pos++)
            sum+=image[im_pos]*temp[x_filt];
        result[res_pos] = sum;
    }
}
```

Figure 8.9: *Code fragment from the `internal_filter` function in the `epic_e` benchmark from Mediabench which exhibits poor performance on ASH.*

Especially for such a tight loop, any overhead operations translate immediately to a slow-down. By tracing the behavior of the code, we discover that at run-time the number of iterations of the innermost loop grows monotonically starting from one. Therefore, the loop has instances with both few and many iterations.

8.2.2.3 `epic_e:internal_filter`

This function is the only kernel from the Mediabench suite which displayed a slowdown in our analysis in Chapter 7. This function is a collection of many nearly identical four-deep nested `for` loops. A fragment is exhibited in Figure 8.9. Let us focus on the innermost two `for` loops. There is very little that can be done to optimize these loops except unrolling the innermost one, which is inefficient due to the non-constant loop bounds. Since the addition involves `double` values, it is not associative so the chain of additions in the unrolled loop cannot be reassociated into a balanced tree.

The run-time behavior of both superscalar and ASH when executing this function is very similar to the one described above for `init_processor`: when the innermost loop has a short trip count the superscalar effectively pipelines the outer loop. When the inner loop has a long trip count, branch prediction is very effective and the critical resource is the floating-point adder. On ASH, unrolling helps when the inner loop has many iterations but harms performance when it has few iterations, both cases occurring quite frequently. We express our gratitude to David Koes, who helped us understand the behavior of this function.

8.2.2.4 Call Strictness (Synthetic Example)

The program in Figure 8.10 exhibits yet another weakness of our current implementation of ASH. In this example the function `unused_arg` ignores its first argument, which is computed using some expensive expression (not shown). In the ASH implementation the call instruction is *strict*, i.e., not lenient — it

```
int
unused_arg(int unused, int used)
{
    return used;
}


int million_calls()
{
    int i, s=0;

    for (i=0; i < 1000000; i++)
        s+=unused_arg(EXPENSIVE(i), i);
    return s;
}
```

Figure 8.10: *Sample code exhibiting performance problems due to the strictness of the call and return instructions.*

waits for all arguments to be available before transferring control to the callee. Therefore, the critical path contains the expensive computation.

The superscalar passes the call arguments in registers and the call instruction is just a branch. The net effect is that the call instruction is *lenient*, since control may be passed to a procedure before all its arguments are computed. In this example the procedure may even return before the expensive computation is terminated because its result is unused.

The `return` instruction behaves similarly. In ASH, it is strict in all of its three arguments — the value returned, the token and the *current execution point*. In contrast, the processor just writes the return value to a register and branches back to the caller. The branch may complete much earlier than the actual register writeback. If the caller ignores the returned value (a frequent occurrence in C), the returned value computation may vanish from the critical path (assuming it does not stall the processor through the use of resources or through the in-order commit stage.)

Unfortunately there are two obstacles towards implementing lenient calls in ASH: (1) the state-machine of the call instruction has to be made more complicated; and (2) strictness is actually required for the *correctness* of the recursive calls because procedure arguments are not steered by "switch" nodes. Non-strict calls could lead to the mixing of data with different generalized iteration indices.

The simplest solution to this problem, not always applicable, is to inline the callee. As explained, in all our experiments we disable inlining, both in the superscalar and in ASH, since the different inlining algorithms in gcc and CASH would produce widely different code — making a comparison of the results very difficult. This policy turns out to be to the advantage of the superscalar.

8.2.2.5 Unoptimized Boolean Computations

Clearly, the use of predicates to steer control-flow at eta operations is an Achilles' heel of ASH. A single CFG cross-edge predicate controls all etas corresponding to that particular edge. If this predicate computation is on the critical path, it slows down the progress of *all* scalar values forwarded by the etas, effectively creating a dynamic barrier.

To compound this problem, examination of the code has revealed that the Boolean computations are not always optimal, despite the use of the sophisticated Espresso Boolean minimization tool (and the sometimes substantial compilation time taken by it, as evidenced in Section 2.7). We have discussed in Section 4.3.5 three reasons for the lack of optimality of the current implementation.

An important related open research question concerns the relationship between leniency and optimality of a Boolean expression. Espresso tries to minimize the depth of a Boolean tree. The underlying assumption is that all inputs are present simultaneously and the signal propagation is minimized. However, if inputs arrive at different times, a deeper tree which takes advantage of leniency and input order arrival might provide the result faster. This subject is an interesting theoretical research question: given some probability distributions on the values of the inputs and on their arrival times, what is the optimal Boolean computation of a given function which minimizes the expected computation time?

8.2.2.6 Inefficient Handling of Recursion

Pegasus assigns all local variables to wires/registers. It never needs to spill them to memory due to lack of available resources. This wastes hardware resources, since scalar values circulate through all hyperblocks where they are live, even if they are not used. This scheme is completely biased against making memory accesses. However, performance suffers when the procedure makes recursive calls, since then *all* scalar value have to be saved and restored. The overhead may be quite substantial. Moreover, some of the values restored may be saved again at another call site, before being ever used, because the stack frames are not cached across several calls as a regular compiler would do. Recursive calls have yet another negative performance effect due to the constraint that they must be placed at the start of a hyperblock. This produces hyperblock fragmentation which reduces the scope for other optimizations. In particular, recursive calls within a loop will break a loop into multiple cycle-free hyperblocks, which are not as well optimized by CASH as a single loop body. The high cost of the current implementation of recursion is one of the main reasons programs such as `130.li` and `134.perl`, which are heavy users of recursion (the hottest functions in these programs are recursive), suffer substantial performance degradations on ASH.

8.2.2.7 Control-Flow Overhead

Another feature of ASH having a detrimental impact on performance is the runtime cost of the machinery used for steering the data flow. This machinery incurs additional costs over a processor for three reasons:

- At “switch”, “mu” and “mux” nodes: the latency of the signal propagation through these nodes is assumed to be logarithmic in the number of inputs. In contrast to processors, in ASH not only branches (i.e., etas) have a cost but also “joins” — either in the form of mu, switch or mux nodes. Although some proposed processor architectures supporting predication may also inject dynamic “multiplexor” micro-operations [WWK$^+$01], in the baseline superscalar that we model, joins are essentially free.

- At the mu-switch construct: as described in Section 6.3.1, the “switch” nodes are controlled by the *current execution point* mu nodes in order to allow an orderly flow of data. However, the net result is that when the eta predicate computation is critical, its criticality is further increased by the mu→switch propagation delay. Control-intensive programs pay a hefty penalty in ASH due to this reason.

- The propagation delay of an eta node is assumed to be one clock cycle (unless its predicate is constant). Therefore in our model, conditional branches always incur the same cost while good prediction can transform branches in a superscalar in operations with an effective latency of zero.

This cost is a more fundamental issue than would appear at first sight. As in any distributed system, global data (or global signals) is replaced with communication and/or synchronization. The steering of data by "eta", "mux", "mu" and "switch" nodes requires coordination between multiple nodes and a broadcast of the branch condition. Even though the cost we model for these operations tends to be relatively low, it still becomes an important overhead in very tight loops, such as the ones exemplified in this section. The token "combine" nodes, as described below, are yet another form of inexpensive-but-nonzero-cost synchronization. Other distributed architectures, such as MIT's Raw, also have to pay for broadcasting the branch conditions to the remote computations [LBF$^+$98].

8.2.2.8 Token Propagation

The LSQ access algorithm described in Section 6.5.3 tends to be quite efficient and, in general, propagates the tokens ahead of the data. However, tokens can be on the critical path due to three reasons:

- Procedure calls are strict and therefore slow-down the token propagation, synchronizing the tokens with the other scalar data.

- Eta nodes controlled by costly predicates slow down the token propagation.

- Finally, even the lowly "combine" operator, which is used to collect tokens to ensure the preservation of run-time dependences, contributes in some cases significantly to the critical path. The cost of a "combine" is assumed to be logarithmic in the number of inputs. Combine operators with a large fan-in, which can arise because of an ambiguous reference or because of a store following a large set of loads, can incur quite a substantial run-time overhead.

The critical path of the hottest function in the `129.compress` benchmark consists mostly of token edges, due to these reasons.

8.3 A Limit-Study with Perfect Branch Prediction

In order to ascertain the importance of branch prediction, we have carried out a limit-study to determine the contribution of control-flow switching predicates (i.e., eta predicates) to the overall program performance. This study consists in running the same measurements as in Section 8.2.1, but with perfect eta predicate prediction.

We have carried out this experiment by modifying the simulator to log all eta-controlling predicate values. Then we have re-run each program replacing the computation of the predicates with values replayed from the log. An eta initiates a log read when it receives its data input. In this way, predicates are never on the critical path.[2] Figure 8.11 shows the performance of the SpecInt95 programs on ASH with perfect predicate prediction. The performance improves somewhat, but still does not match the processor.

[2] This stratagem works as long as the program is fully deterministic and always takes the same execution path at run-time. Certain functions, for example manipulating hashes based on object addresses or depending on random numbers or on time, cannot be replayed in this way.

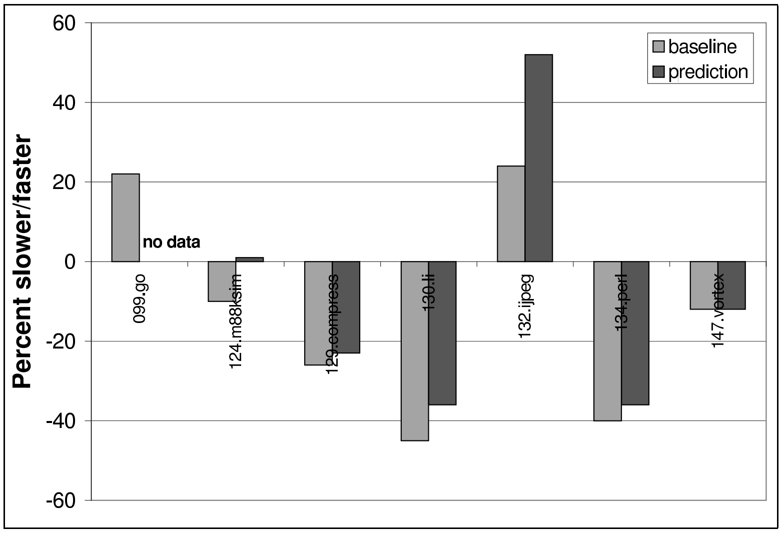


Figure 8.11: *Speed-up of SpecInt95 benchmarks executed on ASH with perfect eta predicate prediction, compared with the original speed-up from Figure 8.1. Simulation of* `099.go` *with predicate log replaying has failed.*

8.4 A Design for Integrating Branch Prediction in Spatial Computation

The comparison to the superscalar has highlighted the importance of branch prediction for high performance in control-intensive programs. Some form of cross-hyperblock branch prediction is a necessary ingredient for unleashing the ILP. In this section we sketch a proposal for integrating a limited form of "branch" prediction and cross-hyperblock speculation into Spatial Computation. This proposal is still just a paper design, but we are fairly confident that it can be implemented without too much effort. A complete implementation and evaluation is an important piece of our planned future work. To our knowledge this design is the first attempt of integrating control speculation into a dataflow model of computation.

Data speculation already exists in Spatial Computation across forward branches — the use of multiplexors and predicate promotion effectively speculates execution along all paths of (most) forward branches. Multiplexors and eta nodes are natural speculation squashing points. The only non-speculative execution points are the eta operations themselves — data is allowed to flow only on the correct path past an eta.

The chief difficulty in extending speculation across hyperblocks (i.e., past etas) is the danger of runaway speculation since there no longer exists a natural squashing point. Another more subtle aspect has to do with the correctness of evaluation: side-effect operations cannot be speculated unless an undo mechanism is put in place, for example by using a transactional cache as in thread-level data speculation [SCM97]. Moreover, speculation could lead to the violation of the execution invariants (described in Section 6.3 and Appendix B) which guarantee program correctness. In particular, there is the danger of "mixing" at the inputs of an operation, both speculative and non-speculative data.

The superscalar processor strongly benefits from its monolithic architecture in implementing speculation and branch prediction. First, it uses global branch prediction tables, which are updated at all branch instructions, and therefore can be used to aggregate path information. Second, the monolithic nature of the processor pipeline, the full dynamic dataflow nature of the register renaming engine (allocating a new register for each produced data), together with the in-order commit stage, allow the internal processor state to be relatively easily flushed on misprediction. The crucial aspect of the architecture is that both the prediction and the flushing mechanism are independent on the actual executed program. In contrast, any Spatial Computation-based scheme will necessarily be distributed and create custom hardware structures for each speculation point.

8.4.1 Compiler-Controlled Dataflow Speculation

Control-flow speculation starts with the compile-time decision of which CFG cross-edges are to be speculatively executed. For each source hyperblock only one outgoing cross-edge can be selected.

The following changes have to be made to the Pegasus building algorithm described in Section 3.3:

1. To each hyperblock h a new predicate input $spec(h)$ is added, which will indicate whether speculation has to be squashed or not. The invariant is $spec(h) =$ "true" if the hyperblock is non-speculative, $spec(h) =$ "false" if it is speculative, and $spec(h)$ is yet undefined if the hyperblock's speculative status is unknown. For an always non-speculative hyperblock, $spec(h)$ is set by the compiler to the constant value "true."

2. Let us consider a cross edge $h_1 \rightarrow h_2$ chosen to become speculative. The predicate controlling the etas forwarding data along this edge are changed from $p(h_1, h_2)$ to constant "true." As a consequence, these etas forward the data irrespective of the actual edge predicate.

3. A new output eta is added for each outgoing cross edge, for initializing $spec(h_2)$. The value forwarded by this eta is $p(h_1, h_2) \wedge spec(h_1)$. This ensures that h_2's speculative status is "true" if and only if $spec(h_1)$ is true (i.e., the source was correctly executed), and execution flows along the $h_1 \rightarrow h_2$ cross-edge.

4. The predicate of each eta in each speculative hyperblock h_2 (i.e., the destination of a speculative cross-edge) is "and"-ed with $spec(h_2)$. Together rules 3 and 4 prevent runaway speculation because data is not forwarded to the next hyperblock unless the non-speculative state of the current hyperblock is confirmed.

5. The predicate of each side-effect operation in h_2 is "and"-ed with $spec(h_2)$. This ensures that the operation gets executed if and only if it is non-speculative.

6. The "return" instruction in a hyperblock h needs to be predicated by $spec(h)$. I.e., "return" is never speculative.

Figure 8.12 illustrates schematically the application of this algorithm on a small program fragment. Hyperblocks 1 and 4 are selected to be non-speculative, therefore their $spec$ flag are constant "true." In hyperblock 3 the predicate of the load has been replaced from p_ld to p_ld $\wedge$ $spec(h_3)$. Note in hyperblock 2 the computation of $spec(3)$ by "and"-ing $spec(2)$ with the original $p(2, 3)$, denoted by $P(2)$ in the figure.

The net run-time effect is that scalar values reaching the head of a speculative cross-edge *always* cross into the next hyperblock when the current hyperblock is certified to be non-speculative, or get squashed otherwise. The use of the "switch" nodes (not shown in this figure) ensures that speculative and non-speculative data cannot get mixed. The first signal reaching the "crt" mu of h_2 will open the gates for all other data signals, coming from the same source. This scheme works for loops as well, by speculating the back-edge.

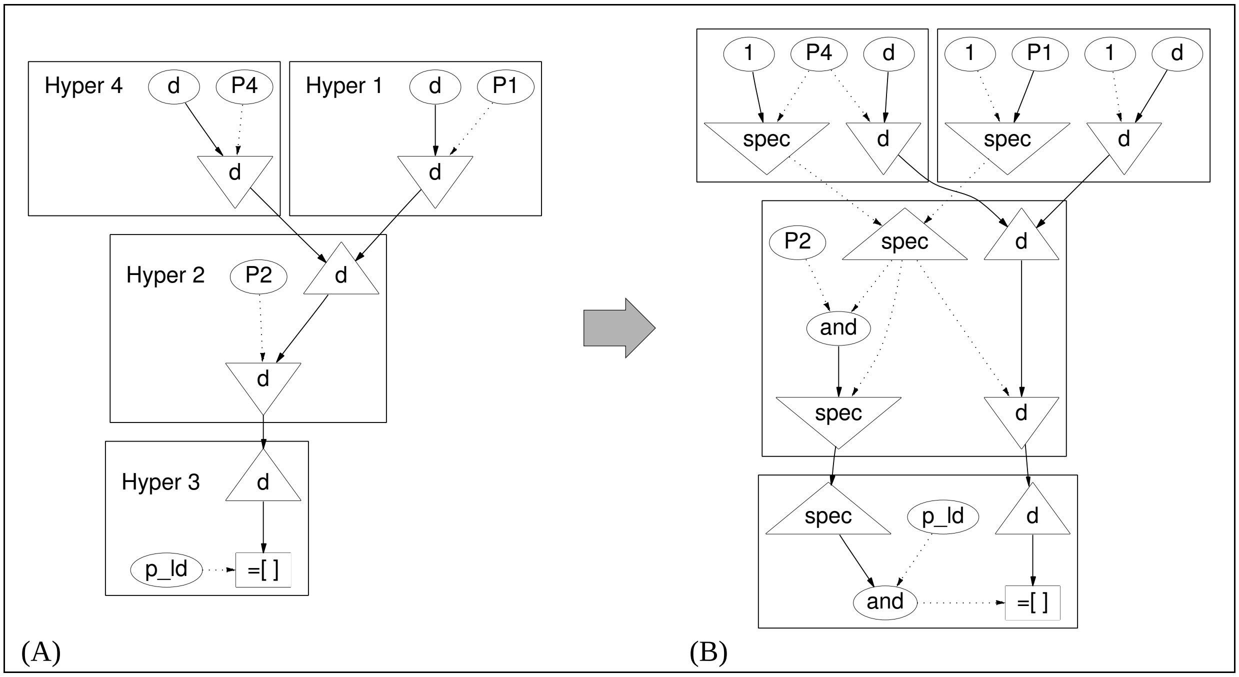


Figure 8.12: *A program fragment comprising four hyperblocks (A) before speculation and (B) after speculating the edges* $1 \rightarrow 2$ *and* $2 \rightarrow 3$*.*

8.4.2 Comments

This scheme works by allowing values to flow forward along some path even if it is uncertain that it is the right path, but by passing along the $spec$ predicate certifying the correctness of the path. The advantage is that all non-side effect operations can be executed early if their inputs are available. At run-time this is an advantage if the computation of $spec$ takes a long time. Scalar computations can proceed ahead regardless of the value of $spec$. By "and"-ing a controlling predicate of a side-effect operation with $spec(h)$ it is ensured that the side-effect never occurs speculatively.

This speculation scheme suffers from some limitations, whose importance still has to be evaluated. First, speculation is compiler-controlled and therefore cannot use run-time information. Second, speculation is allowed only on certain paths. In particular, in order to avoid mixing speculative and non-speculative data, the compiler enforces speculation only *along one execution path*, selected at compilation time. Third, speculative execution is allowed to proceed only *one hyperblock ahead* of the non-speculative wavefront. This prevents run-away speculation. Fourth, side-effect operations are never speculative and they need to wait for the "real" predicate to show up.

The positive side of our proposed scheme is still very much in the nature of ASH — it requires very little extra hardware and features a completely distributed implementation, without using any global control structures. An evaluation of its effectiveness is a subject of future work.

8.5 Conclusions

One interpretation of the results in this section is that they constitute yet another limit study on the amount of instruction-level parallelism available in C programs. Many studies exploring the interplay between architectural features, compilers and ILP have been carried during the course of the years: [LW92, Wal93a, TGH93, RDN93, SBV95, CSY90, SJH89, TF70, NF84, RF72, KMC72, KPG98, UMKK03, SMO03]. Our results add yet another interesting point to the space of possibile architectures. Here are the characterizing features of our study:

- We use realistic compilation technology and realistic memory dependence analysis.
- We use predication and speculation of some of the forward branches.
- We evaluated the impact of no branch prediction and perfect branch prediction.
- We have a virtually unlimited ILP architecture.
- We can model both a realistic and a perfect memory system.
- We model the memory access bandwidth.
- We compile standard benchmarks, large substantial C programs.

From the performance results and from our analysis of some critical code segments, we conclude that in our model of Spatial Computation there are some fundamental limitations towards the exploitation of ILP:

1. Control dependences still limit the ILP, as they do for non-predicated code [LW92].
2. Implementing a generic prediction scheme (be it branch prediction or value prediction) is hindered by the difficulty of building a mechanism for squashing the computation on the wrong paths.
3. The ability of register renaming to remove anti-dependences between the same operation in different iterations is crucial for efficiently executing tight loops.
4. Instruction issue may well be the limiting factor to performance in superscalars [SJH89], since their ability to exploit ILP is excellent.
5. The distributed nature of the computation in ASH requires more remote accesses (i.e., even LSQ accesses become non-local), which are more expensive than in a monolithic system.
6. The presence of even one long-latency non-pipelined operation limits the maximum throughput of the computation of a loop.
7. The cost of the explicit “join” operations in the representation (mu, switch, mux, combine), all of which are essentially “free” in a processor,[3] can sometimes be substantial. This is the cost of *synchronization* which replaces the global signals in monolithic architectures.

This study has increased our admiration for the wonderful capability of superscalar processors of exploiting ILP and pipeline parallelism, through the virtualization[4] of a very limited set of resources: computational units, registers, bandwidth on the internal and external interconnection networks.

[3] The cost of these operations in a processor is essentially paid by the branches.

[4] By virtualization we mean “sharing effectively for the execution of different instructions.”

Chapter 9

Extensions and Future Work

This thesis barely scratches the surface of such an important subject as Spatial Computation. In this section we mention some of the more challenging long-term projects which would extend this research. (We avoid the obvious chores, such as implementing support for the missing C constructs.)

9.1 Building a Complete System

Let us look again at Figure 1.4, which we reproduced here as Figure 9.1. This figure shows that the task of translating C programs into hardware entails creating three different structures: computations, memories and networks. This document only details the making of the computations. In order to make Spatial Computation real, the other two tasks need to be tackled as well.

The network synthesis is currently under development as part of the Verilog back-end of CASH, written by Girish Venkataramani. It is not yet clear how a low-overhead flexible network can be created to accommodate all oddities of C. Function calls thorough pointers, and even return instructions, will require some form of dynamic routing. Even the synthesis of an efficient interconnection network linking circuits to memory poses some very difficult problems.

The aspect of memory partitioning has been tackled the least so far in our work. Some interesting ideas have already been advanced by other research groups: [SSM01, BRM$^+$99, BLAA99, BLAA98]. Unfortunately such algorithms seem to rely on very precise whole-program pointer and dependence analysis and therefore are most likely not scalable to large programs. We expect that the constraint-based algorithms, such as the ones used in CCured to analyze pointers for creating a type-safe version of C [NMW02], can be used to aid pointer analysis and memory partitioning. Another promising avenue is the use of pragmas to guide memory partitioning.

Another important aspect worth investigating is how well the CASH compilation model is amenable to supporting concurrency. Concurrency could be introduced in several ways — either by extending the language with a minimal number of constructs, as in Cilk [BJK$^+$95] by using a library such as `pthreads`, by using a side-effect analysis in the compiler to discover procedures that can be executed concurrently, or by using pragma annotations.

9.2 Hardware Support for ASH

A very important question to investigate is "what is the ideal reconfigurable fabric for supporting ASH." ASH currently assumes an asynchronous hardware substrate. Research in asynchronous logic is still in its

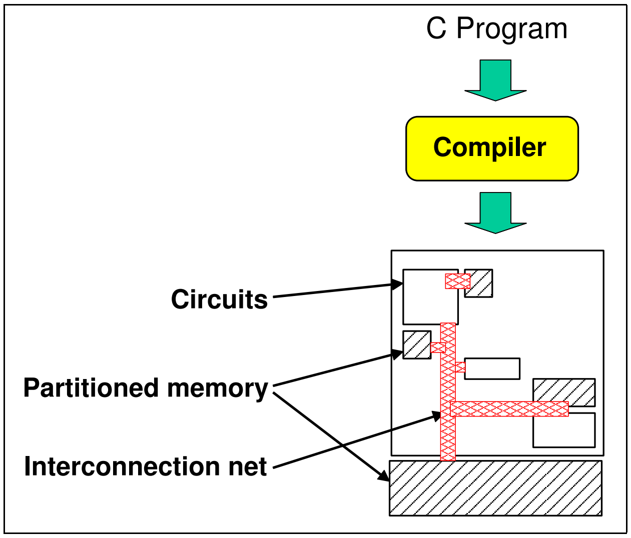


Figure 9.1: *Synthesizing a C program into hardware entails creating (1) circuits for performing the computation; (2) partitioning memory and assigning data; and (3) creating an interconnection network.*

infancy, compared to the much more mature synchronous domain. There are no commercial asynchronous FPGA devices, although devices supporting a form of event notification exist already [BMN+01] and some research prototypes have been proposed, e.g., [TM03]. Most major FPGA circuits operate at a bit-level granularity and incur very large configuration and routing overheads for synthesizing wide computations. A word-oriented architecture will be much more suitable as an implementation target for ASH.

Another hardware support question is whether a *virtualizable* hardware fabric can be created as a support for Spatial Computation, such as our prior work, PipeRench [GSM+99]. This would relieve the compiler and synthesis tools from resource constraints, using run-time reconfiguration to provide the appearance of unlimited hardware resources. We hope that CASH will be a catalyst for attacking the hardware architecture problem, using a philosophy inspired by the RISC approach: build in hardware only the resources that the compiler can use.

9.3 Cross-Breeding ASH and Processors

Finally, there is a lot of cross-fertilization to be attempted between the techniques used for regular processors and ASH. Worthwhile ideas to bring to ASH from processors are:

- Adding support for control-speculation in Spatial Computation, using the scheme detailed in Section 8.4.
- Devising schemes for sharing resources in Spatial Computation, as hinted in Section 6.7.2.4.
- Incorporating profiling information into the compilation process.
- Handling shutdown and exceptions, as suggested in Section 2.5.3.1.

In the other direction, we plan to investigate the addition of architectural/compiler features to regular processors:

- How to communicate dependence information from the compiler to the run-time system. Such information could be used by a clustered load-store queue architecture.
- Architectural support for the new register promotion algorithm from Section 4.6, in the form of hardware-maintained “valid” bits.
- Exploring methodologies for obtaining the benefits of dataflow software pipelining from Section 6.7 on processors without complex software pipelining implementations, by assigning strongly connected components to separate threads in an SMT-like processor [TEE$^+$96].

Finally, another very important research direction is applying formal verification to analyze CASH. This approach may be facilitated by the precise semantics of Pegasus.

Chapter 10

Conclusions

Traditional computer architectures are clearly segmented by the Instruction Set Architecture (ISA), the hardware-software interface definition: compilers and interpreters sit “above”, while the processor hardware is “below”, functioning as an interpreter of the instruction set. In this thesis we have investigated a computational structure which foregoes the existence of an ISA altogether. In this setup the compiler *is* the architecture.

We have investigated one particular instance of an ISA-less architecture, which we dubbed “Spatial Computation.” In Spatial Computation the program is translated directly into an executable hardware form, without using an interpretation layer. The synthesized hardware closely reflects the program structure — the definer-user relationships between program operations are translated directly into point-to-point communication links connecting arithmetic units. The resulting computation engine is completely distributed, featuring no global communication or broadcast, no global register files, no associative structures, and using resource arbitration only for accessing the global monolithic memory. While the use of a monolithic memory does not take advantage of the freedom provided by such an architecture, it substantially simplifies the completely automatic implementation of C language programs. We have chosen to synthesize dynamically self-scheduled circuits, i.e., where computations are carried based on availability of data and not according to a fixed schedule. A natural vehicle for implementing self-scheduled computations is provided by asynchronous hardware.

This thesis has explored both the compile-time and run-time properties of such an architecture. The main contribution of this work with respect to the compilation methodology has been the introduction of a *dataflow intermediate representation* which seamlessly embeds several modern compilation technologies, such as predication, speculation, static-single assignment, an explicit representation of memory dependences, and a precise semantics. We have shown how many classical and several novel optimizations can be efficiently implemented using this representation.

Our investigation in the run-time aspects of Spatial Computation has shown that it can indeed offer superior performance when used to implement programs with resource requirements substantially exceeding the ones available in a traditional CPU. In particular, the execution of multimedia program kernels can achieve very high performance in Spatial Computation, at the expense of very little power. A detailed investigation of the run-time behavior of the spatial structures has also shown that their distributed nature forces them to incur small, but non-negligible overheads when handling control-intensive programs. We have shown that the use of global structures (such as branch prediction, control speculation and register renaming) in traditional superscalar processors can therefore more efficiently execute control-intensive code. Since Spatial Computation complements the capabilities of traditional monolithic processors, a promising avenue of research is the investigation of hybrid computation models, coupling a monolithic and distributed engine.

Chapter 11

Epilogue

In this thesis we have explored the properties of Spatial Computation, a new computational model which translates programs written in high-level languages into hardware static dataflow machines. We have explored both the compilation process for translating ANSI C programs into spatial structures and the run-time properties of this new model of computation.

We will draw a parallel on the contrast between Spatial Computation and today's microprocessors with the approach to operating system architecture, and the controversy between microkernels and monolithic kernels.

- Both Spatial Computation and microkernels break away a relatively monolithic architecture into individual lightweight pieces, well specialized for their particular functionality. Spatial Computation removes global signals and control, in the same way microkernels remove the global address space.

- Communication in Spatial Computation requires handshakes and service invocation in microkernels uses some form of messages or remote procedure call.

- Both Spatial Computation and microkernels build systems out of modular pieces which interact through very precise interfaces, to such a degree that the pieces are interchangeable. For example, in Spatial Computation, one can substitute a pipelined implementation or even a procedure call for an elementary computation, and the correctness of the whole structure is not affected.

- Similar to some microkernels [EKO95], Spatial Computation is very well suited for some particular restricted application domains, namely media processing.

- Spatial Computation is a globally-asynchronous architecture with no global clock. Correspondingly, in a microkernel there is no longer a notion of a single non-preemptive kernel thread of control.

- In Spatial Computation the use of remote services, such as memory accesses or procedure invocations, may require the exchange of complex long-latency messages where a processor uses "global" signals. Similarly, in a microkernel servers must inquire other servers about their internal state instead of just reading global variable values.

- Both monolithic kernels and microprocessors can use global information from different subsystems to make good decisions (for example, a processor can easily implement two-level branch prediction).

- Both monolithic kernels and microprocessors can implement transactional behavior more easily than their modular counterparts. Transaction rollback corresponds in a processor to the squash of wrong-path speculation.

- In both microkernels [Lie96] and Spatial Computation, the cost of remote services is sometimes too high and impacts performance negatively.

- In recent years microkernels (such as Mach [RJO$^+$89]) have been blended with monolithic system architectures to give rise to successful commercial products, such as Mac OS X [App03] and Windows NT [CC92]. Our hope is that, by cross-breeding ideas from Spatial Computation with the monolithic architecture of microprocessors, a more powerful paradigm for computation may emerge.

Appendix A

Benchmarking Methodology

We evaluate Spatial Computation by translating off-the-shelf C programs or program fragments (kernels) into simulated hardware. We use three benchmark suites: Mediabench [LPMS97], integer benchmarks from Spec 95 [Sta95] and benchmarks from Spec 2000 [Sta00]. We use the reference input sets for Spec and the only available input sets for Mediabench. All simulations check the output against the reference to validate correctness. We do not compile or simulate the standard library, for which we do not have the source. In our results we assume that execution of the library is instantaneous. Whenever we compare performance to a processor, we carefully subtract time spent in the standard library from the simulation time of the processor. This is also true for the C constructs we do not support, i.e., `alloca()` and varargs. Whenever a program contains functions we cannot handle, we treat them in the same way as library functions. Unfortunately we also do not simulate the cache for the library routines, and therefore the cache state during our simulations may be incorrect. We hope that the first-order effects are still correct.

While we attempt to present data for all benchmark programs, we do not always succeed. We do not do this in order to hide the negative results [Cit03], but because some of the programs had trouble with our experimental compiler. In particular, here is a list of programs we have failed to compile or execute at all:

- `ghostview` from Mediabench has a lot of script-generated source files and a very complex set of Makefiles which we do not handle.

- `126.gcc` from SpecInt95 does not work with the Suif front-end — parsing and un-parsing generates non-working code.

- `186.crafty` from SpecInt2K uses the non-standard `long long` type and fails to compile with the version of SimpleScalar tools we have (it works very well with our compiler, though).

- `252.eon` from SpecInt2K is a C++ program, which none of our tools handles.

- `253.perlbmk` from SpecInt2K makes a `fork()` system call, which is not supported by any of our simulators.

For most experiments we do not use any of the floating-point programs, either from Spec95 or from Spec2K. We do occasionally give data from SpecFP2K programs, but only for the C and FORTRAN 77 programs. The latter are translated to C using `f2c`. The data for the FORTRAN programs should not be considered very reliable, since the generated C program has completely different aliasing rules compared to FORTRAN, and CASH's memory disambiguator does not take advantage of this fact.

Occasionally compiler or simulator bugs have prevented us from completing the compilation or simulation of some benchmark. In this case we do not present results. We never omit results because they are unfavorable. We present data for all the successful measurements.

Appendix B

Correctness Arguments

In this chapter we argue correctness aspects of Pegasus. Ideally we should show that the representation of a program has exactly the same meaning as the original program. To prove a compiler correct, we should also show that each transformation of the program preserves its meaning. Both these are lofty goals, which are still out of the reach of current research when considered in their full generality, and we will do them very little justice in this thesis. However, we make a few small steps:

- We point towards literature on related subjects that suggests that our representation has very good properties, such as soundness and adequacy.
- We list some important compile-time and run-time invariants of the Pegasus representation.
- Since our representation encompasses some new constructs, such as predication, speculation, and lenient execution, we give informal arguments on the correctness of their use.
- In Section B.4 we actually give a complete, unambiguous precise semantics for each Pegasus operator. This semantics can be used both as a concise reference of the intermediate language (for example, to aid in the implementation of an interpreter), but also as a tool for more serious formal analysis of Pegasus programs.

B.1 Invariants

Not every collection of interconnected Pegasus primitives forms a meaningful computational structure. In this section we enumerate the invariants that are obeyed by any representation of a legal procedure. The algorithm building Pegasus from Section 3.3 should establish these invariants and all optimizations should preserve them. These properties are *necessary* for having a well-formed representation. It is an interesting research question whether they are also *sufficient*.

Of course, these properties do not imply a preservation and a progress theorem because these do not hold for the original C program. In particular, there is no type-safety for the memory accesses.

- All back-edges (i.e., loop-closing edges) originate at an eta and reach a mu (or switch) node.
- All the mu/switch inputs to a hyperblock have the exact same number of input wires. This invariant is true because these wires represent the live variables at a particular program point. If these multiple CFG edges join at that point, there are multiple inputs for each mu node, each corresponding to a different flow of the same variable.

- The eta operators of a hyperblock that go to different destination hyperblocks have disjoint predicates. This implies that control always flows to a single successor hyperblock.
- The logical union of all eta predicates in a hyperblock is “true”, indicating that execution always reaches a next hyperblock (a *progress* property). This implies that any hyperblock must end in etas or a “return.”
- All etas corresponding to a given cross-edge should always have the exact same controlling predicate. This means that all live values at a program point “flow” in the same direction at run-time.
- All inputs of an n-ary operation originate at some other operation. An operation cannot have only some of the inputs connected.
- The predicates controlling a mux are always disjoint. Their union does not necessarily amount to “true.” This invariant is true because a value can flow to a mux along a single path and mux control predicates correspond to the last edge on the path.
- The graph is “strongly typed”, i.e., the wire types always match the output type of the producer of the value and the input types of *all* consumers of the value carried by the wire. This ensures a *type preservation* property of the computation.
- All side-effecting operations (except loads provable from immutable values) must have a token input and a token output connected to other side-effecting operations.
- In any “cut” of the Pegasus graph the union of the token edges crossing the cut in one direction must “guard” all mutable memory locations. This ensures that no two non-commuting side-effect operations (i.e., on different sides of the cut) can be (incorrectly) reordered.
- There are no edges flowing “around” a recursive call (i.e., if there is a path $a \rightarrow^* c \rightarrow^* b$, where c is a recursive call, all paths $a \rightarrow^* b$ must contain c). This ensures that values from two different recursive invocations of a same procedure cannot be mixed. This invariant is established by the rules for synthesizing recursive calls in Section 3.3.11.

Some important run-time invariants have been described in Section 6.3, in relationship to the “generalized iteration index” introduced there:

- For any operation, all inputs that contribute to the computation of the output have the same index.
- The consecutive values flowing on every edge have strictly increasing indices.

These invariants are naturally maintained for acyclic code regions. For these invariants to also be maintained for programs with loops, the use of switches to steer control-flow is sufficient, as described in Section 6.3.1.

B.2 Properties

Pegasus is a restricted form of synchronous dataflow computation. The dataflow model of computation has been introduced by Karp and Miller in 1996 [KM66], and the synchronous dataflow by Lee and Messerschmitt [LM87]. By its very structure and semantics, a well-formed Pegasus graph has the following properties:

Determinism: (Church-Rosser): A Pegasus graph will produce the same result regardless of the order and duration of the computation of each node (assuming each node is deterministic). This property was proved for a simple scalar graph in the Karp-Miller paper. The extension to memory-access operations can be argued observing that the use of token edges guarantees the preservation of Bernstein's conditions [Ber66] for deterministic execution of a parallel computation.

Deadlock: The fact that a Pegasus computation cannot deadlock follows from some of the invariants described above. The Pegasus graph of a single hyperblock is a restricted form of synchronous dataflow, in which every node consumes and produces exactly one token. The arguments of Lee and Messerschmitt can be used to prove deadlock impossibility for each hyperblock in isolation. The invariants regarding eta nodes and the inter-hyperblock connections can be used to argue the lack of deadlock for a complete procedure-level Pegasus graph.

Adequacy: The adequacy of a representation is the property that programs with isomorphic program dependence graphs produce the same results. A proof of adequacy for a type of dependence graph can be found in [HPR88]. Hopefully, these results can also be extended to apply to Pegasus.

B.3 Speculative Computation Gives Correct Results

As discussed in Section 3.3, Pegasus uses a method equivalent to the *predicate promotion* technique described in [MLC$^+$92], which removes predicates from some instructions or replaces them with weaker predicates, enabling their speculative execution. If the hyperblock code is in static-single assignment form, we can prove that every instruction with no side-effects can be safely and completely promoted to be executed unconditionally.

Here is a proof sketch. We can distinguish four types of instructions: (a) instructions with side-effects (i.e., memory accesses and procedure calls); (b) predicate computations; (c) multiplexors; and (d) all other instructions.

The instructions of type (a) cannot be executed speculatively. We will argue below that instructions of type (b) (i.e., the predicates) can be speculatively computed. It follows that the multiplexors (c) always make the correct choices because they select based on the predicates. As a conclusion, instructions of type (d) can always be safely executed speculatively because they have no side-effects and their inputs are always correct.

We now prove that all instructions of type (b), which compute predicate values, can be speculatively executed. The key observation is that, although the predicates are themselves speculatively computed, their value is always correct. We will illustrate the proof using a simple example.

Consider the CFG fragment in Figure B.1. According to the definition of predicate computations from Section 3.3, $P(c) = (P(a) \wedge B(a,c)) \vee (P(b) \wedge B(b,c))$. Let us assume that during an execution of the program, basic block a is executed and branches to c. This means that block b is not executed. Therefore, the branch condition $B(b,c)$ may have an incorrect value (for instance, because block b changes some values which influence the branch computation). However, by using induction on the depth of the block, we can assume that $P(b)$ has the correct value, "false." Therefore, the value of $B(b,c)$ is irrelevant for the computation of $P(c)$.

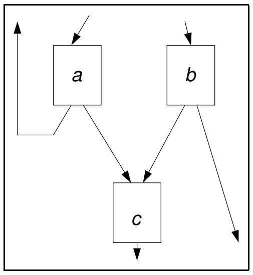


Figure B.1: *Fragment of a control-flow graph.*

B.4 Formal Semantics

In this section we describe precisely the meaning of the basic constructs of Pegasus. We use a Plotkin-style operational semantics [Plo81].

We denote the set of program values by V. (These are the values manipulated by the actual program: integers, floating point numbers, etc.) To this set we add some distinguished special values:

- the Booleans $\mathbf{T}$ and $\mathbf{F}$
- an abstract token value τ
- a *don't care* value $\triangle$
- a special "undefined" value, denoted by $\perp$
- *node names* are also values in order to implement "call" and "return" operations — the `call` must name the parameter nodes of the called procedure, while the `return` indicates the `call continuation` node to which control is returned[1]
- a $\mathbf{sent}$ value, used for implementing lenient evaluation (Section 6.4)

A state associates to each edge $e \in E$ a value in V. That is, a state is a mapping $\sigma : E \to V$. We denote the fact that edge e has value v by $\sigma(e) = v$. A computation is defined by a state transition. We describe the semantics of each operation by indicating: (a) when the computation can be carried and (b) how it changes the state. We denote by

$$op \qquad \frac{\textit{precond}}{\textit{change}}$$

the fact that operation *op* requires precondition *precond* to be executed and then incurs the *change* modification to the state. The transition relation is nondeterministic: at some point several operations may be eligible for execution and the semantics does not specify the execution order. It is expected that well-formed programs have a confluence property, i.e., the result is the same regardless of the execution order.

We write $\sigma' = \sigma[e \mapsto v]$ to indicate a state σ' which coincides with σ, except that it maps e to v. If node N has inedges i_1 and i_2 and outedge o, we write this as $o = N(i_1, i_2)$. We use $addr$ to denote values which represent memory addresses. We write `update`$(addr, v)$ to indicate that value v is stored in memory address $addr$, and `lookup`$(addr)$ for the contents of memory location $addr$.

[1] To simplify the presentation we do not use call continuation nodes.

We use suggestive edge names: p denotes predicates, t tokens, i and o input and respectively output edges, pc edges whose values denote node names (from Program Counter). For example, $\sigma(pc) = \text{main}$ indicates that the value on the pc edge is the identity of the node representing the "main" procedure. In this case, we denote by $\sigma(pc).arg_1$ the first argument of "main", $\sigma(pc).t$ the token argument of "main", etc.

For simplicity we allow in this model only a fanout of one for all nodes, except a special "fanout" node, which can have an unlimited number of outputs and which copies the input to all the outputs.

We model constants as having the *current execution point* crt as an input.

In the initial state all values are undefined, i.e., $\forall e.\sigma(e) = \bot$.

We abbreviate $\sigma(e) \neq \bot$ with def(e), (i.e., the edge e has a "defined" value) and erase(a) as a shorthand for $[a \mapsto \bot]$ (i.e., a state in which a becomes undefined). We abuse the notation and use def$()$ and erase$()$ with multiple arguments.

$$o = \text{unary\_op}(i) \quad \frac{\text{def}(i), \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \text{unary\_op}(\sigma(i))] \circ \text{erase}(i)}$$

$$o = \text{binary\_op}(i_1, i_2) \quad \frac{\text{def}(i_1, i_2), \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \text{binary\_op}(\sigma(i_1), \sigma(i_2))] \circ \text{erase}(i_1, i_2)}$$

$$(o_1, \ldots, o_n) = \text{fanout}(i) \quad \frac{\text{def}(i), \neg\text{def}(o_1, \ldots, o_n)}{\sigma' = \sigma[o_1 \mapsto \sigma(i)][\ldots][o_n \mapsto \sigma(i)] \circ \text{erase}(i)}$$

$$o = \text{constant}(crt) \quad \frac{\neg\text{def}(o), \text{def}(crt)}{\sigma' = \sigma[o \mapsto \text{constant}] \circ \text{erase}(crt)}$$

All constants have *crt* as input

$$o = \text{uninitialized}(crt) \quad \frac{\neg\text{def}(o), \text{def}(crt)}{\sigma' = \sigma[o \mapsto \text{uninitialized}] \circ \text{erase}(crt)}$$

$$o = \text{hold(i, p)} \quad \frac{\text{def}(i)}{\sigma' = \sigma}$$

Do not ack this input.

$$o = \text{hold(i, p)} \quad \frac{\text{def}(i), \sigma(p) = \mathbf{T}}{\sigma' = \sigma[o \mapsto \sigma(i)] \circ \text{erase}(p)}$$

$$o = \text{hold(i, p)} \quad \frac{\text{def}(p), \text{def}(i), \sigma(p) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \sigma(i)] \circ \text{erase}(p, i)}$$

Ack both inputs only now.

$$(o, crtl) = \text{mu}(i_1, \ldots, i_n) \quad \frac{\exists k.\text{def}(i_k), \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \sigma(i_k), ctrl \mapsto k] \circ \text{erase}(i_k)}$$

Exactly one of the inputs of a *mu* can be present at a time.

$$o = \text{switch}(ctrl, i_1, \ldots, i_n) \quad \frac{\text{def}(i_k), \sigma(crtl) = k, \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \sigma(i_k)] \circ \text{erase}(i_k, ctrl)}$$

$$o = \text{eta}(p, i) \quad \frac{\sigma(p) = \mathbf{T}, \text{def}(i), \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \sigma(i)] \circ \text{erase}(i, p)}$$

$$o = \text{eta}(p, i) \quad \frac{\sigma(p) = \mathbf{F}, \text{def}(i)}{\sigma' = \sigma \circ \text{erase}(i, p)}$$

$$t_0 = \text{V}(t_1, \ldots, t_n) \quad \frac{\neg\text{def}(t_0), \text{def}(t_1, \ldots, t_n)}{\sigma' = \sigma[t_0 \mapsto \tau] \circ \text{erase}(t_1, \ldots, t_n)}$$

$$(t, a) = \text{frame(n)}(t_0) \quad \frac{\text{def}(t_0), \neg\text{def}(t, a)}{\sigma' = \sigma[t \mapsto \tau, a \mapsto \text{allocate n bytes}] \circ \text{erase}(t_0)}$$

$$t = \text{pop(n)}(t_0) \quad \frac{\text{def}(t_0), \neg\text{def}(t)}{\sigma' = \sigma[t \mapsto \tau] \circ \text{erase}(t_0), \text{deallocate last frame}}$$

$$a = \text{sp}(t) \quad \frac{\text{def}(t), \neg\text{def}(a)}{\sigma' = \sigma[a \mapsto \text{last frame allocated}] \circ \text{erase}(t)}$$

$$t = \text{G(n)}(t_0, p) \; \frac{\text{def}(t_0)}{\text{counter++}, \sigma' = \sigma \circ \text{erase}(t_0)}$$

Token generator: increment internal counter originally initialized to n.

$$t = \text{G(n)}(t_0, p) \; \frac{\sigma(p) = \mathbf{T}, \text{counter} > 0}{\text{counter}--, \sigma' = \sigma[t \mapsto \tau] \circ \text{erase}(p)}$$

$$t = \text{G(n)}(t_0, p) \; \frac{\sigma(p) = \mathbf{T}, \text{counter} \leq 0}{\text{counter}--, \sigma' = \sigma \circ \text{erase}(p)}$$

$$t = \text{G(n)}(t_0, p) \; \frac{\sigma(p) = \mathbf{F}, \text{counter} \neq \text{n}}{\sigma' = \sigma}$$

Do not ack p.

$$t = \text{G(n)}(t_0, p) \; \frac{\sigma(p) = \mathbf{F}, \text{counter} = \text{n}}{\sigma' = \sigma \circ \text{erase}(p)}$$

Restore G to original state.

$$o = \text{mux}(i_1, p_1, \dots, i_n, p_n) \; \frac{\neg\text{def}(o), \text{def}(i_1.p_1, \dots i_n, p_n), \exists k.\sigma(p_k) = \mathbf{T}}{\sigma' = \sigma[o \mapsto \sigma(i_k)] \circ \text{erase}(i_1, p_1, \dots, i_n, p_n)}$$

At most one predicate of a mux can be $\mathbf{T}$.

$$o = \text{mux}(i_1, p_1, \dots, i_n, p_n) \; \frac{\neg\text{def}(o), \text{def}(i_1.p_1, \dots i_n, p_n), \forall k.\sigma(p_k) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \triangle] \circ \text{erase}(i_1, p_1, \dots, i_n, p_n)}$$

$$t_0 = \text{store}(\mathit{addr}, p, v, t) \; \frac{\neg\text{def}(t_o), \text{def}(\mathit{addr}, v, t), \sigma(p) = \mathbf{T}}{\sigma' = \sigma[t_0 \mapsto \tau] \circ \text{erase}(\mathit{addr}, p, t, v), \texttt{update}(\sigma(\mathit{addr}), \sigma(v))}$$

$$t_0 = \text{store}(\mathit{addr}, p, v, t) \; \frac{\neg\text{def}(t_o), \text{def}(\mathit{addr}, v, t), \sigma(p) = \mathbf{F}}{\sigma' = \sigma[t_0 \mapsto \tau] \circ \text{erase}(\mathit{addr}, p, t, v)}$$

$$(o, t_0) = \text{load}(\mathit{addr}, p, t) \; \frac{\neg\text{def}(o, t_0), \sigma(p) = \mathbf{T}, \text{def}(t, \mathit{addr})}{\sigma' = \sigma[o \mapsto \texttt{lookup}(\mathit{addr})][t_0 \mapsto \tau] \circ \text{erase}(\mathit{addr}, p, t)}$$

$$(o, t_0) = \text{load}(\mathit{addr}, p, t) \; \frac{\neg\text{def}(o, t_0), \sigma(p) = \mathbf{F}, \text{def}(t, \mathit{addr})}{\sigma' = \sigma[o \mapsto \triangle][t_0 \mapsto \tau] \circ \text{erase}(\mathit{addr}, p, t)}$$

$$o = \text{arg} \; \frac{\neg\text{def}(o), \text{def}(\text{arg})}{\sigma' = \sigma[o \mapsto \sigma(\text{arg})] \circ \text{erase}(\text{arg})}$$

$$\text{return}(i, t, \mathit{pc}, \mathit{crt}) \; \frac{\text{def}(i, t, \mathit{pc})}{\sigma' = \sigma[\sigma(\mathit{pc}).o \mapsto \sigma(i)][\sigma(\mathit{pc}).t \mapsto \tau][\sigma(\mathit{pc}).\mathit{crt} \mapsto \tau] \circ \text{erase}(i, t, p, \mathit{crt})}$$

$$(o, t_0, \mathit{crt}) = \text{call}_k(t, p, \mathit{pc}, \mathit{crt}_0, i_1, \dots, i_n) \; \frac{\text{def}(\mathit{pc}, t, \mathit{crt}_0, i_1, \dots, i_n), \sigma(p) = \mathbf{T}}{\begin{array}{c} \sigma' = \sigma[\sigma(\mathit{pc}).\mathit{arg}_1 \mapsto \sigma(i_1)][\dots][\sigma(\mathit{pc}).\mathit{arg}_n \mapsto \sigma(i_n)] \\ \circ[\sigma(\mathit{pc}).t \mapsto \tau][\sigma(\mathit{pc}).\mathit{pcin} \mapsto \text{call}_k] \\ [\sigma(\mathit{pc}).\mathit{crt} \mapsto \tau] \circ \text{erase}(i_1, \dots, i_n, \mathit{crt}_0, t, p, \mathit{pc}) \end{array}}$$

$$(o, t_0, \mathit{crt}) = \text{call}_k(t, p, \mathit{pc}, \mathit{crt}_0, i_1, \dots, i_n) \; \frac{\text{def}(\mathit{pc}, t, i_1, \dots, i_n), \sigma(p) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \triangle][t_0 \mapsto \tau][\mathit{crt} \mapsto \tau] \circ \text{erase}(i_1, \dots, i_n, t, p, \mathit{pc}, \mathit{crt}_0)}$$

Semantics for lenient evaluation We present only formulas for some input arrival orders. The others are completely symmetric. An internal value $\mathbf{sent}_o$ is used to remember whether the output for the current "generalized iteration" has been already emitted (the output may be emitted before all inputs are available, so the an "output busy" may mean either that the output has been emitted already, or that the prior output has not been acknowledged yet).

$$o = \text{or}(i_1, i_2) \; \frac{\sigma(i_1) = \mathbf{T}, \neg\text{def}(o), \neg\text{def}(\mathbf{sent}_o)}{\sigma' = \sigma[o \mapsto \mathbf{T}][\mathbf{sent}_o \mapsto \mathbf{T}]}$$

Emit output, do not ack input.

$$o = \text{or}(i_1, i_2) \; \frac{\text{def}(i_1, i_2), \sigma(\mathbf{sent}_o) = \mathbf{T}}{\sigma' = \sigma \circ \text{erase}(i_1, i_2, \mathbf{sent}_o)}$$

$$(o, t_0) = \text{load}(addr, p, t) \; \frac{\neg\text{def}(o, t_0), \sigma(p) = \mathbf{T}, \text{def}(t, addr)}{\sigma' = \sigma[o \mapsto \texttt{lookup}(addr)][t_0 \mapsto \tau] \circ \text{erase}(addr, p, t)}$$

$$(o, t_0) = \text{load}(addr, p, t) \; \frac{\neg\text{def}(o), \sigma(p) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \triangle][\mathbf{sent}_0 \mapsto \mathbf{T}]}$$

Emit data early.

$$(o, t_0) = \text{load}(addr, p, t) \; \frac{\neg\text{def}(t_0), \sigma(p) = \mathbf{F}, \text{def}(t)}{\sigma' = \sigma[t_0 \mapsto \tau][\mathbf{sent}_t \mapsto \mathbf{T}]}$$

Emit output token if token input present.

$$(o, t_0) = \text{load}(addr, p, t) \; \frac{\sigma(p) = \mathbf{F}, \text{def}(t, addr), \mathbf{sent}_o = \mathbf{T}, \mathbf{sent}_t = \mathbf{T}}{\sigma' = \sigma \circ \text{erase}(addr, p, t, \mathbf{sent}_o, \mathbf{sent}_t)}$$

Ack inputs.

Similar rules for stores, procedure calls and all other predicated operations.

$$o = \text{mux}(i_1, p_1, i_2, p_2) \; \frac{\neg\text{def}(o), \text{def}(i_1), \sigma(p_1) = \mathbf{T}, \neg\text{def}(\mathbf{sent}_o)}{\sigma' = \sigma[o \mapsto \sigma(i_1)][\mathbf{sent}_o \mapsto \mathbf{T}]}$$

$$o = \text{mux}(i_1, p_1, i_2, p_2) \; \frac{\text{def}(i_1, p_1, i_2, p_2), \sigma(\mathbf{sent}_o) = \mathbf{T}}{\sigma' = \sigma \circ \text{erase}(i_1, i_2, p_1, p_2, \mathbf{sent}_o)}$$

$$o = \text{mux}(i_1, p_1, i_2, p_2) \; \frac{\forall k . \sigma(p_k) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \triangle][\sigma(\mathbf{sent}_o) = \mathbf{T}]}$$

$$o = \text{mux}(i_1, p_1, i_2, p_2) \; \frac{\text{def}(i_1, p_1, i_2, p_2), \mathbf{sent}_o = \mathbf{T}}{\sigma' = \sigma \circ \text{erase}(i_1, i_2, p_1, p_2, \mathbf{sent}_o)}$$

$$o = \text{mux}(i_1, p_1, i_2, p_2) \; \frac{\text{def}(i_1, i_2), \sigma(p_1) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \sigma(i_2)][\mathbf{sent}_o \mapsto \mathbf{T}]}$$

Special optimization.

Bibliography

[AC01] Marnix Arnold and Henk Corporaal. Designing domain-specific processors. In *International Symposium on Hardware/Software Codesign (CODES)*, pages 61–66. ACM Press, 2001.

[ACM$^+$98] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, and Wen mei W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 227–237, June 1998.

[AFKT03] Alex Aiken, Jeff Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[AJLA95] Vicki H. Allan, Reese B. Jones, Randal M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[AKPW83] R. Allen, Kennedy K., C. Porterfield, and J.D. Warren. Conversion of control dependence to data dependence. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 177–189, Austin, Texas, January 1983.

[AMKB00] V. Agarwal, H.S. Murukkathampoondi, S.W. Keckler, and D.C. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *International Symposium on Computer Architecture (ISCA)*, June 2000.

[AmWHM97] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *International Symposium on Computer Architecture (ISCA)*, December 1997.

[AN90] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers (TOC)*, 39(3), 1990. Also published as MIT Computations Structures Group Technical Memo, Memo-271, 1987.

[App98] Andrew W. Appel. SSA is functional programming. ACM SIGPLAN Notices, April 1998.

[App03] Apple Computer Inc. *Mac OS X Kernel Programming*. Apple Computer, Inc., 2003.

[Arn99] Guido Arnout. C for system level design. In *Design, Automation and Test in Europe (DATE)*, pages 384–387, Munich, Germany, March 1999.

[AS93] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[ASP+99] D.L. August, J.W. Sias, J.-M. Puiatti, S.A. Mahlke, D.A. Connors, K.M. Crozier, and W.-M.W. Hwu. The program decision logic approach to predicated execution. In *International Symposium on Computer Architecture (ISCA)*, pages 208–219, 1999.

[Atk02] Darren C. Atkinson. Call graph extraction in the presence of function pointers. In *the 2002 International Conference on Software Engineering Research and Practice*, June 2002.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. In *Computer Architecture News*, volume 25, pages 13–25. ACM SIGARCH, June 1997.

[BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 159–170, June 1994.

[Ber66] A. J. Bernstein. Program analysis for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15:757–762, October 1966.

[BG02a] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, Montpellier (La Grande-Motte), France, September 2002.

[BG02b] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[BG03a] C.F. Brej and J.D. Garside. Early output logic using anti-tokens. In *International Workshop on Logic Synthesis*, pages 302–309, May 2003.

[BG03b] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 216–227, San Francisco, CA, March 23-26 2003.

[BGS00] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, British Columbia, June 2000.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 207–216, Santa Barbara, California, July 19-21 1995.

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to data flow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[BLAA98] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Memory bank disambiguation using modulo unrolling for Raw machines. In *International Conference on High Performance Computing*, Chennai, India, December 1998.

[BLAA99] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Maps: A compiler-managed memory system for Raw machines. In *International Symposium on Computer Architecture (ISCA)*, Atlanta, GA, 1999.

[BMBG02] Mihai Budiu, Mahim Mishra, Ashwin Bharambe, and Seth Copen Goldstein. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–66, Napa Valley, CA, April 2002.

[BMN$^+$01] V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP - a self-reconfigurable data processing architecture. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2001)*, Las Vegas, June 2001.

[BRM$^+$99] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1999.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers (TOC)*, C-35(8):677–691, August 1986.

[BSVHM84] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and C. McMullin. *Logic Minimization Algorithms for Digital Circuits*. Kluwer Academic Publishers, Boston, MA, 1984.

[BSWG00] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *European Conference on Parallel Processing (EUROPAR)*, volume 1900 of *Lecture Notes in Computer Science*, pages 969–979, München, Germany, 2000. Springer Verlag.

[CA88] David E. Culler and Arvind. Resource requirements of dataflow programs. In *International Symposium on Computer Architecture (ISCA)*, pages 141–150, 1988.

[Cal02] Timothy John Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*. PhD thesis, Computer Science Department of the University of California at Berkeley, 2002.

[CC92] Helen Custer and David N. Cutler. *Inside Windows NT*. Microsoft Press, Redmond, WA, November 1992.

[CCK90] S. Carr, D. Callahan, and K. Kennedy. Improving register allocation for subscripted variables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, White Plains NY, June 1990.

[CFR$^+$91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[CG93] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 36–45. ACM Press, 1993.

[Cit03] Daniel Citron. MisSPECulation: Partial and misleading use of SPEC2000 in computer architecture conferences. In *Panel Session of the International Symposium on Computer Architecture*, San Diego, CA, June 2003. Position Paper.

[CK94] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software — Practice and Experience*, 24(1), January 1994.

[CL03] Jean-Francois Collard and Daniel Lavery. Optimizations to prevent cache penalties for the Intel Itanium 2 processor. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, March 23-26 2003.

[Cla99] T.A.C.M. Claasen. High speed: not the only way to exploit the intrinsic computational power of silicon. In *IEEE International Solid-State Circuits Conference*, pages 22–25, San Francisco, CA, 1999. IEEE Catalog Number: 99CH36278.

[CLL+96] Fred Chow, Raymond Lo, Shin-Ming Liu, Sun Chan, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *International Conference on Compiler Construction (CC)*, pages 253–257, April 1996.

[CMS96] S. Carr, Q. Mangus, and P. Sweany. An experimental evaluation of the sufficiency of scalar replacement algorithms. Technical Report TR96-04, Michigan Technological University, Department of Computer Science, 1996.

[CNBE03] Tiberiu Chelcea, Steven M. Nowick, Andrew Bardsley, and Doug Edwards. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. submitted to IEEE Transactions on Computer Aided Design, 2003.

[Coo00] John Cooley. DAC'00 trip report: ”the READ ME file DAC” — or — ”113 engineers review DAC 2000 in Los Angeles, CA, June 5-9, 2000”. http://www.deepchip.com/posts/dac00.html, 2000.

[Cor03] Celoxica Corporation. Handel-C language reference manual, 2003.

[CoW00] CoWare, Inc. Flexible platform-based design with the CoWare N2C design system, October 2000.

[CSC+99] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated static single assignment. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[CSC+00] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[CSY90] Ding-Kai Chen, Hong-Hen Su, and Pen-Chung Yew. The impact of synchronization and granularity in parallel systems. In *International Symposium on Computer Architecture (ISCA)*, pages 239–248, 1990.

[CTM02] Nathan Clark, Wilkin Tang, and Scott Mahlke. Automatically generating custom instruction set extensions. In *Workshop on Application-Specific Processors (WASP)*, 2002.

[CW95] Steve Carr and Qunyan Wu. The performance of scalar replacement on the HP 715/50. Technical Report TR95-02, Michigan Technological University, Department of Computer Science, 1995.

[CW98] Timothy J. Callahan and John Wawrzynek. Instruction level parallelism for reconfigurable computing. In Hartenstein and Keevallik, editors, *International Conference on Field Programmable Logic and Applications (FPL)*, volume 1482 of *Lecture Notes in Computer Science*, Tallinin, Estonia, September 1998. Springer-Verlag.

[CW02] João M. P. Cardoso and Markus Weinhardt. PXPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. In *International Conference on Field Programmable Logic and Applications (FPL)*, Montpellier (La Grande-Motte), France, September 2002.

[CX02] Keith D. Cooper and Li Xu. An efficient static analysis algorithm to detect redundant memory operations. In *Workshop on Memory Systems Performance (MSP '02)*, Berlin, Germany, June 2002.

[Den74] Jack B. Dennis. First version of a data flow procedure language. In *Lecture Notes in Computer Science 19: Programming Symposium*, pages 362–376. Springer-Verlag: Berlin, New York, 1974.

[DHP$^+$01] Pedro Diniz, Mary Hall, Joonseok Park, Byoungro So, and Heidi Ziegler. Bridging the gap between compilation and synthesis in the DEFACTO system. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2001.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technical University, Eindhoven, the Netherlands, 1965.

[DM98] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *25 Years of the International Symposia on Computer Architecture (selected papers)*, pages 125–131. ACM Press, 1998.

[DZC$^+$02] W. R. Davis, N. Zhang, K. Camera, D. Markovic, T. Smilkstein, M. J. Ammer, E. Yeo, S. Augsburger, B. Nikolic, and R. W. Brodersen. A design environment for high throughput, low power dedicated signal processing systems. *IEEE Journal of Solid-State Circuits*, 37(3):420–431, March 2002.

[ECF$^+$97] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping applications to the RaPiD configurable architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1997.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, 1995.

[Esp00] Jorge Ernesto Carrillo Esparza. Evaluation of the OneChip reconfigurable processor. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2000.

[FAT02] J. Foster, A. Aiken, and T. Terauchi. Flow-sensitive type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 2002.

[FBD02] Brian Fields, Rastislav Bodík, and Mark D.Hill. Slack: Maximizing performance under technological constraints. In *International Symposium on Computer Architecture (ISCA),* pages 47–58, 2002.

[FBF^{+}00] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *International Symposium on Computer Architecture (ISCA),* pages 203–213. ACM Press, 2000.

[Fra70] Dennis J. Frailey. Expression optimization using unary complement operators. In *Proceedings of a symposium on Compiler optimization,* pages 67–85, 1970.

[FRB01] Brian A Fields, Shai Rubin, and Rastislav Bodík. Focusing processor policies via critical-path prediction. In *International Symposium on Computer Architecture (ISCA),* 2001.

[Gal95] David Mark Gallagher. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1995.

[Gao86] Guang R. Gao. *A Pipelined Code Mapping Scheme for Static Data Flow Computers*. PhD thesis, MIT Laboratory for Computer Science, 1986.

[Gao90] Guang R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining*. Kluwer Academic Publishers, December 1990.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24 (6):547–578, 1996.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 121–133, San Diego, California, January 1998.

[GJJS96] D.M. Gillies, D.R. Ju, R. Johnson, and M. Schlansker. Global predicate analysis and its application to register allocation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO),* pages 114–125, 1996.

[GJLS87] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[GKL99] A. Ghosh, J. Kunkel, and S. Liao. Hardware synthesis from C/C++. In *Design, Automation and Test in Europe (DATE)*, pages 384–387, Munich, Germany, March 1999.

[GM94] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* pages 61–72. ACM Press, 1994.

[GM95] M. Gokhale and A. Marks. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays. In W. Moore and W. Luk, editors, *International Conference on Field Programmable Logic and Applications (FPL),* pages 399–408, Oxford, England, August 1995. Springer.

[GN99] Emden Gansner and Stephen North. An open graph visualization system and its applications to software engineering. *Software Practice And Experience*, 1(5), 1999. http://www.research.att.com/sw/tools/graphviz.

[GP89] Guang R. Gao and Z. Paraskevas. Compiling for dataflow software pipelining. Technical Report MCGS 89-11, Computer Science Department, McGill University, 1989.

[GSAK00] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 49–56, 2000.

[GSD$^+$02] Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, Alex Nicolau, Timothy Kam, Michael Kishinevsky, and Shai Rotem. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *Design Automation Conference (DAC)*, pages 898–903. ACM Press, 2002.

[GSK$^+$01] Sumit Gupta, Nick Savoiu, Sunwoo Kim, Nikil D. Dutt, Rajesh K. Gupta, and Alexandru Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Design Automation Conference (DAC)*, pages 269–272, 2001.

[GSM$^+$99] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: a coprocessor for streaming multimedia acceleration. In *International Symposium on Computer Architecture (ISCA)*, pages 28–39, Atlanta, GA, 1999.

[Gup92] Rajiv Gupta. Generalized dominators and post-dominators. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 246–257, Albuquerque, New Mexico, January 1992.

[GWN91] Guang R. Gao, Yue-Bong Wong, and Qi Ning. A timed Petri-net model for fine-grain loop scheduling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 204–218. ACM Press, 1991.

[HA00] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *IEEE/ACM International Conference on Computer-aided design (ICCAD)*, San Jose, California, November 2000.

[Hav93] Paul Havlak. Construction of thinned gated single-assignment form. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 768 of *Lecture Notes in Computer Science*, pages 477–499, Portland, Ore., 1993. Berlin: Springer Verlag.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61. ACM Press, 2001.

[HMH01] R. Ho, K. Mai, and M. Horowitz. The future of wires. *the IEEE*, 89(4):490–504, April 2001.

[HPR88] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 146–157, 1988.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[Ins] European Telecommunications Standards Institute. GSM enhanced full rate specifications 06.51, 06.60-63 and 06.82.

[itr99] International technology roadmap for semiconductors (ITRS). http://public.itrs.net/Files/1999_SIA_Roadmap/Design.pdf, 1999.

[JBP$^+$02] A. Jones, D. Bagchi, S. Pal, X. Tang, A. Choudhary, and P. Banerjee. Pact hdl: A c compiler targeting asics and fpgas with power and performance optimizations. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Grenoble, France, October 2002.

[JLP91] Richard Johnson, Wei Li, and Keshav Pingali. An executable representation of distance and direction. *Languages and Compilers for Parallel Computers*, 4, 1991.

[Joh99] Doug Johnson. Programming a Xilinx FPGA in “C”. *Xcell Quarterly Journal*, 34, 1999.

[Joh00] Doug Johnson. Architectural synthesis from behavioral C code to implementation in a Xilinx FPGA. *Xcell Journal*, 2000.

[JP93] R. Johnson and K. Pingali. Dependence-based program analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 78–89, Albuquerque, New Mexico, June 1993.

[KBVG03] David Koes, Mihai Budiu, Girish Venkataramani, and Seth Copen Goldstein. Programmer specified pointer independence. Technical Report CMU-CS-03-123, Carnegie Mellon University, Department of Computer Science, April 2003.

[KC98] Hwayong Kim and Kiyoung Choi. Transformation from C to synthesizable VHDL. In *Asia Pacific Conference on Hardware Description Languages (APCHDL)*, July 1998.

[KCJ00] J. Knoop, J.-F. Collard, and R. D. Ju. Partial redundancy elimination on predicated code. In *International Static Analysis Symposium (SAS)*, number 1824 in Lecture Notes in Computer Science, pages 260 – 279, Santa Barbara, CA, June 2000. Springer-Verlag, Heidelberg.

[KKP$^+$81] David J. Kuck, R. H. Kuhn, David A. Padua, B. Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–218, Williamsburg, VA, January 1981.

[KM66] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966.

[KMC72] D.J. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers (TOC)*, C-21:1293–1310, 1972.

[KNY^{+}99] Andrew Kay, Toshio Nomura, Akihisa Yamada, Koichi Nishida, Ryoji Sakurai, and Takashi Kambe. Hardware synthesis with Bach system. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, Orlando, 1999.

[KPG98] Artur Klauser, Abhijit Paithankar, and Dirk Grunwald. Selective eager execution on the PolyPath architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 250–259, June 1998.

[LBF^{+}98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 46–57, 1998.

[LC97] John Lu and Keith D. Cooper. Register promotion in C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319. ACM Press, 1997.

[LCD^{+}00] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Design Automation Conference (DAC)*, 2000.

[LCD02] Jong-eun Lee, Kiyoung Choi, and Nikil Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In *IEEE/ACM International Conference on Computer-aided design (ICCAD)*, pages 649–654. ACM Press, 2002.

[LCK^{+}98] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37. ACM Press, 1998.

[lD99] C level Design. System Compiler: Compiling ANSI C/C++ to synthesis-ready HDL. Whitepaper, 1999.

[LFK^{+}93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1), January 1993.

[LH98] Christopher Lapkowski and Laurie J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In *the 1998 International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 128–143, March 1998.

[Lie96] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[Lin95] Andrew Matthew Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, Computer Science Department, 1995. CS-TR-95-21.

[LM87] E.A. Lee and D.A. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers (TOC)*, C36 (1):24–35, January 1987.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–335, 1997.

[LS93] I. Lemke and G. Sander. Visualization of compiler graphs. Technical Report Design report D 3.12.1-1, USAAR-1025-visual, ESPRIT Project #5399 Compare, Universität des Saarlandes, 1993.

[LS99] Luciano Lavagno and Ellen Sentovich. ECL: A specification environment for system-level design. In *Design Automation Conference (DAC)*, pages 511–516, New Orleans, LA, June 1999.

[LTG97] Stan Liao, Steven W. K. Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Design Automation Conference (DAC)*, pages 70–75, 1997.

[LVL03] Marisa López-Vallejo and Juan Carlos López. On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Transactions Design Automation Electronic Systems*, 8(3):269–297, 2003.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *International Symposium on Computer Architecture (ISCA)*, 1992.

[MB59] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *International Symposium on the Theory of Switching Functions*, pages 204–243, 1959.

[MH00] Tsutomu Maruyama and Tsutomu Hoshino. A C to HDL compiler for pipeline processing on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2000.

[MHM$^+$95] Scott A. Mahlke, Richard E. Hauk, James E. McCormick, David I. August, and Wen mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *International Symposium on Computer Architecture (ISCA)*, pages 138–149, Santa Margherita Ligure, Italy, May 1995. ACM.

[Mic99] Giovanni De Micheli. Hardware synthesis from C/C++ models. In *Design, Automation and Test in Europe (DATE)*, Munich, Germany, 1999.

[MLC$^+$92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Computer Architecture (ISCA)*, pages 45–54, Dec 1992.

[MNP$^+$03] Alain J. Martin, Mika Nystrm, Karl Papadantonakis, Paul I. Penzes, Piyush Prakash, Catherine G. Wong, Jonathan Chang, Kevin S. Ko, Benjamin Lee, Elaine Ou, James Pugh, Eino-Ville Talvala, James T. Tong, and Ahmet Tura. The Lutonium: A sub-nanojoule asynchronous 8051 microcontroller. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, May 2003.

[MPJ$^+$00] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: A modular reconfigurable architecture. In *International Symposium on Computer Architecture (ISCA)*, June 2000.

[MRS+01] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth sensitive code generation in a custom embedded accelerator design system. In *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, St. Goar, Germany, March 2001.

[Muc97] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc, 1997.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[Nel81] B. J. Nelson. Remote procedure call. Technical Report CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

[NF84] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers (TOC)*, C-33(11):968–976, November 1984.

[Nic89] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers (TOC)*, 38 (5):664–678, 1989.

[NMW02] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[OBM90] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 257–271, 1990.

[PBJ+91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *ACM Symposium on Principles of Programming Languages (POPL)*, volume 18, 1991.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Springer Verlag, editor, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, 1998.

[RC03] Davide Rizzo and Osvaldo Colavin. A scalable wide-issue clustered VLIW with a reconfigurable interconnect. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, CA, 2003.

[RCP+01] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. Najjar, and A.P.W. Böhm. An automated process for compiling dataflow graphs into hardware. *IEEE Transactions on VLSI*, 9 (1), February 2001.

[RDK$^+$98] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 1998.

[RDN93] Lawrence Rauchwerger, Pradeep K. Dubey, and Ravi Nair. Measuring limits of parallelism and characterizing its vulnerability to resource constraints. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–117. IEEE Computer Society Press, 1993.

[RF72] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers (TOC)*, C-21(12):1405–1411, December 1972.

[RGLS96] John C. Ruttenberg, Guang R. Gao, Woody Lichtenstein, and Artour Stoutchinin. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, 1996.

[RJO$^+$89] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, and Michael B. Jones. Mach: a system software kernel. In *IEEE Computer Society International Conference: Technologies for the Information Superhighway*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[Ron82] Gao Guang Rong. An implementation scheme for array operations in static data flow computers. Technical Report MIT-LCS-TR-280, MIT Laboratory for Computer Science, May 1982.

[RR99] Ray Roth and Dinesh Ramanathan. A high-level design methodology using C++. In *IEEE International High Level Design Validation and Test Workshop*, November 1999.

[RR00] Ray Roth and Dinesh Ramanathansed. Enabling HW/SW codesign using Cynlib. In *Design, Automation and Test in Europe (DATE)*, March 2000.

[RS94] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmed functional units. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 172–180, November 1994.

[RTT01] Robert B. Reese, Mitch A. Thornton, and Cherrice Traver. Arithmetic logic circuits using self-timed bit level dataflow and early evaluation. In *International Conference on Computer Design (ICCD)*, page 18, Austin, TX, September 23-26 2001.

[SA02] Mukund Sivaraman and Shail Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 35–42. ACM Press, 2002.

[SAJ96] Andrew Shaw, Arvind, and R. Paul Johnson. Performance tuning scientific codes for dataflow execution. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1996. Also MIT Computation Structures Group Memo 381.

[SAmWH00] John W. Sias, David I. August, and Wen mei W. Hwu. Accurate and efficient predicate analysis with binary decision diagrams. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2000.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture (ISCA)*, volume 23 (2), Santa Margherita Ligure, Italy, May 1995. ACM.

[SCM97] J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Architectural support for thread-level data speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[SGR$^+$01] R. Schreiber, S. Aditya (Gupta), B.R. Rau, S. Mahlke, V. Kathail, B. Ra. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 2001.

[SJ98] A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on SSA form. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–25. ACM Press, 1998.

[SJH89] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 290–302. ACM Press, 1989.

[SKA94] Michael Schlansker, Vinod Kathail, and Sadun Anik. Height reduction of control recurrences for ILP processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 40–51, 1994.

[SMO03] Steven Swanson, Ken Michelson, and Mark Oskin. WaveScalar. Technical Report 2003-01-01, Washington University at Seattle, Computer Science Department, January 2003.

[SN00] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, Eilat, Israel, April 2000.

[SNBK01] K. Sankaralingam, R. Nagarajan, D.C. Burger, and S.W. Keckler. A technology-scalable architecture for fast clocks and high ILP. In *Workshop on the Interaction of Compilers and Computer Architecture*, January 2001.

[Som] Fabio Somenzi. *CUDD: CU Decision Diagram Package*. Department of Electrical and Computer Engineering, University of Colorado at Boulder, release 2.3.1 edition.

[SP98] Donald Soderman and Yuri Panchul. Implementing C algorithms in reconfigurable hardware using C2Verilog. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 339–342, Los Alamitos, CA, April 1998. IEEE Computer Society Press.

[SRV98] V. Srinivasan, S. Radhakrishnan, and R. Vemuri. Hardware/software partitioning with integrated hardware design space exploration. In *Design, Automation and Test in Europe (DATE)*, pages 28–35. IEEE Computer Society, 1998.

[SS95] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *the IEEE*, 83(12):1609–1624, 1995.

[SSC01] Greg Snider, Barry Shackleford, and Richard J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 115–124. ACM Press, 2001.

[SSL$^+$92] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, University of California at Berkeley, May 1992.

[SSM01] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Transactions on VLSI*, 2001.

[Sta95] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[Sta00] Standard Performance Evaluation Corp. *SPEC CPU 2000 Benchmark Suite*, 2000. http://www.specbench.org/osg/cpu2000.

[Ste95] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations*, pages 62–70, 1995.

[Sut89] Ivan Sutherland. Micropipelines: Turing award lecture. *Communications of the ACM*, 32 (6):720–738, June 1989.

[SVR$^+$98] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens. A programming environment for the design of complex high speed ASICs. In *Design Automation Conference (DAC)*, pages 315–320, San Francisco, June 1998.

[Syn] Synopsis, Inc. SystemC Cocentric compiler. http://www.synopsys.com/products/cocentric_systemC/cocentric

[Tar72] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TEE$^+$96] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *International Symposium on Computer Architecture (ISCA)*, pages 191–202, Philadelphia, PA, May 1996.

[Ten] Inc. Tensilica. http://www.tensilica.com.

[TF70] G.S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers (TOC)*, C-19:885–895, October 1970.

[TGH93] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. The effects of resource limitations on program parallelism. Advanced Compilers, Architectures and Parallel Systems (ACAPS) Technical Memo 52, January 1993.

[TM03] John Teifel and Rajit Manohar. Programmable asynchronous pipeline arrays. In *International Conference on Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, September 2003.

[Tom67] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

[TP95] Peng Tu and David Padua. Efficient building and placing of gating functions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 47 – 55, 1995.

[TS99] Hervé Touati and Mark Shand. PamDC: a C++ library for the simulation and generation of Xilinx FPGA designs. http://research.compaq.com/SRC/pamette/PamDC.pdf, 1999.

[TSI$^+$99] A. Takayama, Y. Shibata, K. Iwai, H. Miyazaki, K. Higure, and X.-P. Ling. Implementation and evaluation of the compiler for WASMII, a virtual hardware system. In *International Workshop on Parallel Processing*, pages 346–351, 1999.

[TVVA03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David August. Compiler optimization-space exploration. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, March 23-26 2003.

[UMKK03] A. K. Uht, D. Morano, A. Khalafi, and D. R. Kaeli. Levo - a scalable processor with high IPC. *Journal of Instruction-Level Parallelism*, 5, August 2003.

[VBG03] Girish Venkataramani, Mihai Budiu, and Seth Copen Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. Submitted for review to Design Automation Conference., December 2003.

[vN45] John von Neumann. First draft of a report on the EDVAC. Contract No. W-670-ORD-492, Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. Reprinted (in part) in Randell, Brian. 1982. Origins of Digital Computers: Selected Papers, Springer-Verlag, Berlin Heidelberg, June 1945.

[VvdB90] A. H. Veen and R. van den Born. The RC compiler for the DTN dataflow computer. *Journal of Parallel and Distributed Computing*, 10:319–332, 1990.

[Wak99] Kazutoshi Wakabayashi. C-based synthesis experiences with a behavior synthesizer, Cyber. In *Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 1999.

[Wal93a] David W. Wall. Limits of instruction-level parallelism. Technical Report Digital WRL Research Report 93/6, Digital Western Research Laboratory, November 1993.

[WAL$^+$93b] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II compiler and architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, Napa Valley, CA, Apr 1993.

[WC96] R. D. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In J. Arnold and K. L. Pocek, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 126–135, Napa, CA, April 1996.

[WCES94] D. Weise, R. F. Crew, M. Ernst, and B Steensgaard. Value Dependence Graphs: Representation without taxation. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–310, January, 1994.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 83–94. ACM Press, 1980.

[WFW$^+$94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

[WH95] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In P. Athanas and K. L. Pocek, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 99–107, Napa, CA, April 1995.

[Whe01] David A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size. http://www.dwheeler.com/sloc, November 2001.

[Wil90] Ted Williams. Latency and throughput tradeoffs in self-timed speed-independent pipelines and rings. Technical Report CSL-TR-90-431, Stanford University, Computer Systems Laboratory, August 1990.

[Wir98] Niklaus Wirth. Hardware compilation: Translating programs into circuits. *IEEE Computer*, 31 (6):25–31, June 1998.

[WKMR01] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Design Automation Conference (DAC)*, Las Vegas, Nevada, June 2001.

[WO00] Kazutoshi Wakabayashi and Takumi Okamoto. C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. *IEEE Transactions on Computer-Aided Design*, 19(12):1507–1522, December 2000.

[Wol92] Michael Wolfe. Beyond induction variables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 27 (7), pages 162–174, New York, NY, 1992. ACM Press.

[WWK$^+$01] Perry H. Wang, Hong Wang, Ralph M. Kling, Kalpana Ramakrishnan, and John P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *International Symposium on High-Performance Computer Architecture (HPCA)*, Nuevo Leone, Mexico, January 2001.

[YLR$^+$03] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* San Diego, CA, June 2003.

[YMHB00] Alex Zhi Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable unit. In *International Symposium on Computer Architecture (ISCA)*, ACM Computer Architecture News. ACM Press, 2000.

[YSB00] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. A C compiler for a processor with a reconfigurable functional unit. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Northwestern University, April 2000.

[ZB02] Ning Zhang and Bob Brodersen. The cost of flexibility in systems on a chip design for signal processing applications . http://bwrc.eecs.berkeley.edu/Classes/EE225C/Papers/arch_design.doc, Spring 2002.

[ZRG+02] Paul S. Zuchowski, Christopher B. Reynolds, Richard J. Grupp, Shelly G. Davis, Brendan Cremen, and Bill Troxel. A hybrid ASIC and FPGA architecture. In *IEEE/ACM International Conference on Computer-aided design (ICCAD)*, pages 187–194. ACM Press, 2002.

[ZWM+01] Y. Zhao, A. Wang, M. Moskewicz, C. Madigan, and S. Malik. Matching architecture to application via configurable processors: A case study with the Boolean satisfiability problem. In *International Conference on Computer Design (ICCD)*, 2001.